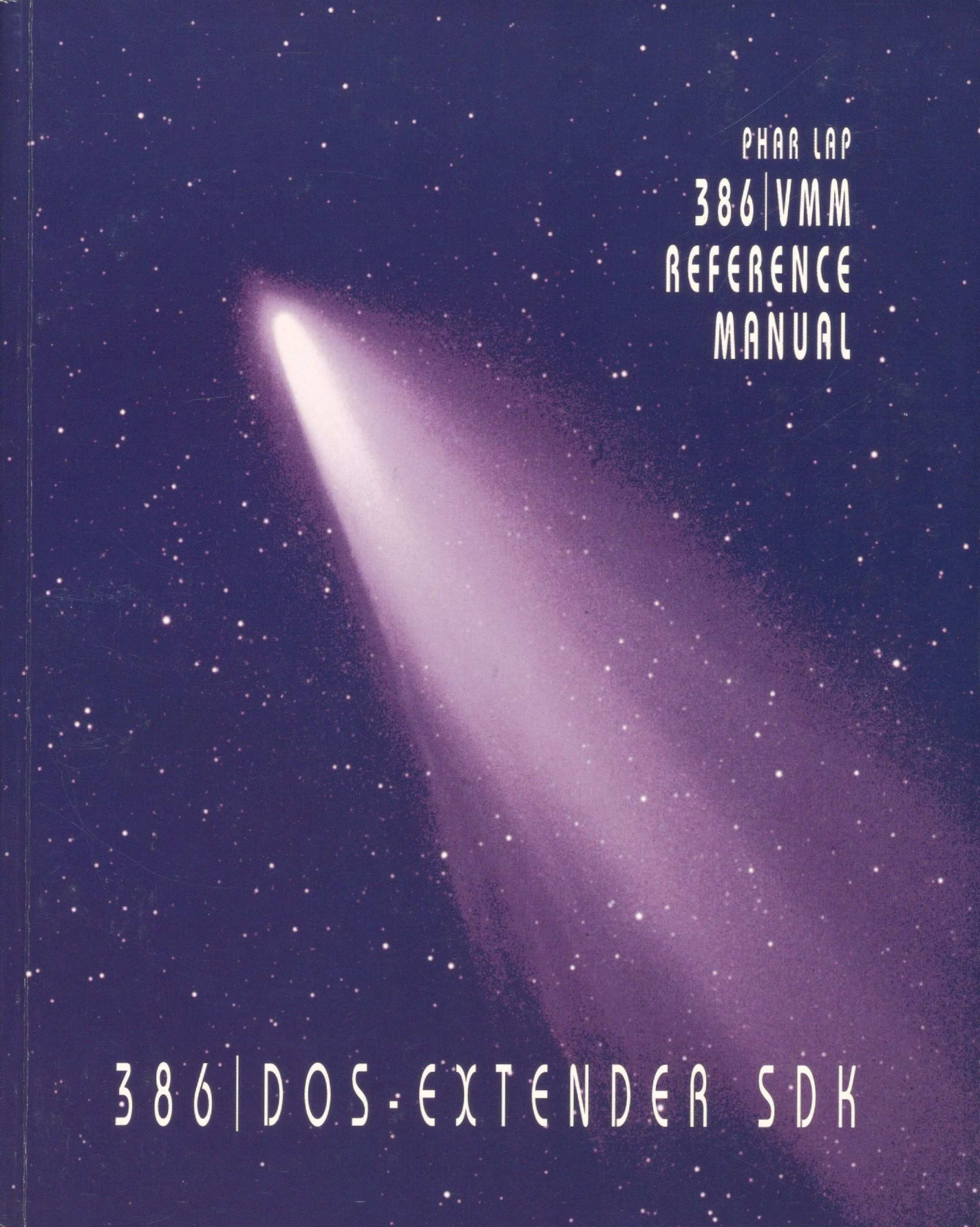

PHAR LAP
386|VMM
REFERENCE
MANUAL
386|DOS-EXTENDER SDK

386 | VMM Reference Manual

Phar Lap Software, Inc.
60 Aberdeen Avenue, Cambridge, MA. 02138
(617) 661-1510, FAX (617) 876-2972
dox@pharlap.com
tech-support@pharlap.com

First edition: April 1989.
Second edition: January 1991.

Table of Contents

Preface

This manual is a technical reference describing the enhancements to the 386|DOS-Extender environment provided by 386|VMM. Readers are assumed to be familiar with the material contained in the *386|DOS-Extender Reference Manual.* For most applications, the information contained in Sections 2.1 and 2.2 of this manual is all that is needed to bring up an application with virtual memory. The remainder of the manual should be used as a reference, or as a guide for tuning program performance by systems programmers familiar with the operation of virtual memory systems.

This manual contains numerous references to 386|DOS-Extender command line switches and system calls which are documented in the *386|DOS-Extender Reference Manual.* There are also a number of 386|DOS-Extender switches and system calls which relate directly to operation with 386|VMM. The 386|VMM-specific command line switches are documented in Section 2.3 of this manual. The 386|VMM-specific system calls are documented in Appendix C.

Throughout this manual, numbers ending in the letter "h" are given in hexadecimal (e.g., 100h = 256 decimal). All other numbers are given in decimal (e.g., 1024). The term "conventional memory" is used to refer to memory below one MB. It is memory that can be directly accessed when executing in the real mode of the 80386. The term "extended memory" is used to refer to memory above one MB.

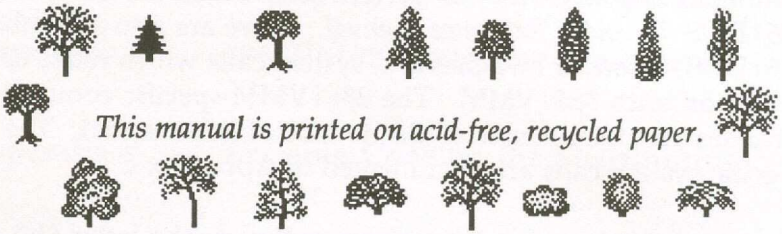
Manual Conventions

This manual relies on certain conventions to convey certain types of information. On the following pages, these are the conventions:

`courier`	indicates the items on the command line.
`italics`	indicate the items on the command line that must have a user-entered name or value in place of the items in italics.
`[ ]`	square brackets indicate that one or more of the enclosed items may be chosen. It is valid not to choose any of the items in square brackets.

Related Documentation and Books

Phar Lap Software, Inc., *386|DOS-Extender Reference Manual.*



If, after you have read this book, you find that you have suggestions, improvements or comments that can make this a better manual, please call us, write us or send us mail at:

Phar Lap Software, Inc.
60 Aberdeen Avenue, Cambridge, MA. 02138
(617) 661-1510, FAX (617) 876-2972
dox@pharlap.com
tech-support@pharlap.com

This manual was produced on a Macintosh IIcx, using MS-Word. The examples illustrated in this manual were developed with 386|VMM, version 3.0.

Chapter 1

Introducing 386 | VMM

386 | VMM provides a demand–paged virtual memory environment for programs running under 386 | DOS–Extender. Programs running under 386 | VMM can have a memory space larger than the available physical memory of the computer. Unused portions of the program's code and data are paged to disk and are automatically brought into memory as they are needed. This functionality permits programs with large memory requirements to run on computers configured with relatively small amounts of memory.

386 | VMM is distributed as an add–in driver for 386 | DOS–Extender. When it is loaded into memory by 386 | DOS–Extender during initialization, 386 | VMM installs handlers for the page fault processor exception of the 80386 chip and for the timer tick hardware interrupt. These handlers, plus services provided by the 386 | DOS–Extender environment, enable 386 | VMM to implement demand–paged virtual memory. 386 | VMM also provides services used by 386 | DOS–Extender to enhance its memory management capabilities.

There are two versions of 386 | VMM: a development version and a runtime version. The development version is provided for in-house use by software developers, and is distributed as a file named VMMDRV.EXP. The runtime version can be bound into an application for resale and is provided when a runtime license is purchased. It is distributed as a file named VMMDRVB.EXP. From the point of view of the application program, there is no difference between the two versions. The only difference is the way the two versions are identified for loading when an application is run under 386 | DOS–Extender.

386 | VMM runs in any hardware and software environment supported by 386 | DOS–Extender. This includes all 80386, 80386SX, and 80486-based hardware systems compatible with the IBM PC AT, IBM PS/2 Micro Channel, and EISA architectures. In addition to running under DOS,

386|VMM and 386|DOS-Extender are compatible with Microsoft Windows 3.0 in real and standard modes, Quarterdeck DESQview 386, and all EMS emulators that support the VCPI interface.

Note: 386|VMM is always disabled under DPMI. The host should always provide virtual memory in a DPMI environment.

Chapter 2

Using 386 | VMM

This chapter provides the basic information needed to bring up an application with virtual memory. The first section, Building Programs for Virtual Memory, describes controlling the size of the program's initial address space when it is loaded. This section also provides information on dynamic heap allocation with selected popular compilers. The second section, Command Line Syntax, covers the syntax for both the development and runtime versions of 386 | VMM. Section 2.3, Command Line Switches, provides information on the 386 | DOS-Extender command line switches that apply specifically to operation with virtual memory, as follows: virtual memory driver; swap file location; page replacement policy; policies for increasing the swap file; paging out of .EXP file; locking program stack; creating a page fault log file.

2.1 Building Programs for Virtual Memory

Programs which run under 386 | VMM are written and built in much the same way as programs which run under 386 | DOS–Extender without virtual memory. If the program has any hardware interrupt handlers, they must be locked in memory so no page faults can occur during a hardware interrupt (please see Section 3.7). The program must also have reasonable memory allocation behavior: memory should be allocated on an as-needed basis, rather than attempting to allocate all available memory, since "all available memory" means all RAM memory plus all unused disk space under virtual memory. Memory allocation under virtual memory is discussed in Section 3.2.

It is important to control the size of the program's initial address space when it is loaded. Normally, a virtual memory program should only get its minimum memory requirements when it is loaded, and should dynamically allocate additional virtual memory as it is needed.

At load time the program is allocated enough memory to contain the code and data in the .EXP file, plus some extra memory controlled with the -MINDATA and -MAXDATA switches (which are specified when the program is linked). The -MINDATA value specifies the minimum amount of extra memory which must be allocated to the program. The -MAXDATA value specifies the maximum amount of extra memory to be allocated if possible. When the program first gets control, the amount of extra memory allocated to it is always between MINDATA and MAXDATA bytes. This extra memory is typically used in high level languages like C as a memory heap, from which the program can allocate and free memory by calling runtime library routines provided with the compiler.

Programs which run without virtual memory typically are built without the -MINDATA and -MAXDATA switches, so the default values assigned by the linker for these parameters are used. Zero is the default value for MINDATA, and four gigabytes is the default for MAXDATA (i.e., the maximum amount possible). Thus, when a non-virtual memory program is loaded, it gets all the available physical memory on the machine. If there is a lot of memory available, it gets a large memory heap. If there is not much memory, it gets a very small heap, or cannot be loaded at all because there is not enough memory to meet the minimum required for the program.

When a program runs under 386 | VMM, the size of the memory space that it sees is no longer limited by the physical memory on the machine, but instead can grow almost arbitrarily large. This means that the program will load successfully even if its minimum memory requirements are larger than available memory. It is unreasonable to give a program a four-gigabyte virtual address space when it is initially loaded (because this would require an unreasonably large swap file), so the MAXDATA value is interpreted differently under 386 | VMM. The MAXDATA value is used only if additional physical memory is available after the program's minimum memory requirements are met. If there is additional physical memory, the program's address space is only increased by the amount of memory available. Thus, a small virtual memory program linked with the default MAXDATA value, and loaded on a computer with a lot of memory, sees an initial address space equal to the available physical memory. A program whose minimum memory requirements are larger than available memory (a common situation,

since that is the purpose of using virtual memory) will initially get a memory heap of MINDATA bytes, which, as mentioned above, is zero by default.

Since most programs need to use a memory heap, some provision must be made to ensure the program's runtime memory needs are met. For most compliers, such as compilers from MetaWare, Watcom, SVS, and MicroWay, the memory allocation routines dynamically increase the size of the program's virtual address space when it runs out of space in the memory heap. If the runtime library supports this, the program should be linked with the -MINDATA and -MAXDATA switches both set to zero, and the program's runtime memory needs are automatically met.

If the compiler runtime library **cannot** support dynamic memory allocation, then the -MINDATA switch must be used at link time to guarantee that the program obtains a heap large enough to satisfy its runtime requirements. The -MAXDATA switch should be given the same value as the -MINDATA switch.

The sections below discuss dynamic heap allocation with some popular compilers.

2.1.1 MetaWare High C and Professional Pascal Compilers

The MetaWare High C and Professional Pascal 80386 compilers both support dynamic heap allocation. Version 1.5 and later of High C, and version 2.8 and later of Professional Pascal, have this feature enabled by default. The default configuration of earlier versions of these compilers, however, disables dynamic heap allocation. To enable dynamic heap allocation with version 1.4 of High C or version 2.7 of Professional Pascal, it is necessary to modify the MetaWare initializer, which is distributed with the compilers as a file named INIT.ASM. Near the top of the file is a line containing a symbol definition which is commented out:

```
; PHAR_LAP_CAN_GROW_HEAP equ 1
```

Modify this line and remove the comment character, so that it looks like:

```
PHAR_LAP_CAN_GROW_HEAP equ 1
```

Assemble the modified INIT.ASM file with this command:

```
386asm init -include \hc386\lib\src\
```

The -INCLUDE switch specifies the directory which contains some header files included by INIT.ASM. The directory path is dependent on where the compiler files are installed on the hard disk.

This command creates the file INIT.OBJ, which is then linked to the application program, replacing the standard version of INIT.OBJ, which is in the C or Pascal runtime library. An example link string is:

```
386link myprog init -lib \hc386\hce -maxdata 0
```

Alternatively, the 386|LIB librarian can be used to replace the INIT.OBJ module in the runtime library. Once it is installed in the library, the new module is automatically retrieved from the library at link time and need not be explicitly specified on the command line when the program is linked. Replace the module in the library with this command:

```
386lib \hc386\hce -replace init
```

Any application linked with the modified INIT.ASM and running under 386|VMM can obtain almost unlimited amounts of memory by calling the runtime library memory allocation routine.

The distribution disk for 386|VMM contains an example virtual memory C program called VM.C which allocates a large memory buffer and scans it to cause paging. The batch file VMBLD.BAT, also on the distribution disk, demonstrates how to build the program using the MetaWare High C compiler. Please see Appendix B for a complete list of example programs included on the distribution disk.

2.1.2 Watcom C/386 and Fortran 77/386 Compilers

All versions of the Watcom C/386 and Fortran 77/386 compliers support dynamic heap allocation with 386|VMM. There is no need for any special edits to files; the support is built in.

2.1.3 SVS 386/C, 386/FORTRAN–77, and SVS 386/Pascal Compilers

All versions of the SVS 386/C, 386/FORTRAN–77, and SVS 386/Pascal compilers automatically support dynamic heap allocation with 386|VMM.

2.1.4 Microway Fortran–386 and C–386 Compilers

Microway Fortran–386 and C–386 versions 2.0 and later automatically support dynamic heap allocation with 386|VMM. For some versions, Microway sold non-VM and VM compilers separately; if you're not sure whether your copy supports 386|VMM, check the compiler version and serial number with Microway.

2.1.5 Assembly Language Programs

Programs written in assembly language typically allocate their own memory. As with high level languages, the most desirable option is to enhance the memory allocator so that when no memory is available, more is obtained by increasing the program's virtual memory space. This is done by using the Resize Segment system call, INT 21h function 4Ah, to increase the size of the program's data segment. Please see the *386|DOS–Extender Reference Manual*. If the maximum amount of memory the program will ever need to allocate is known, then that amount can be specified at link time with the -MINDATA switch, instead of modifying the program's memory allocator to support dynamic memory allocation.

2.2 Command Line Syntax

The command line syntax to run application programs under 386|VMM takes two forms: for the development version and for the runtime version. This is the only difference between the development and runtime versions of 386|VMM. The following sections describe the command line syntax for each version.

2.2.1 Development Version

The 386|VMM development version is distributed as a file named VMMDRV.EXP. When the application program is run under 386|DOS-Extender or one of the Phar Lap debuggers, the -VMFILE command line switch (please see Section 2.3.1) specifies the name and location of the virtual memory driver file. If no file name extension is given, the default extension .EXP is assumed. For example, to run the protected mode application program BIGSORT.EXP with virtual memory, type the following command:

run386 -vmfile \pharlap\vmmdrv bigsort

2.2.2 RUNTIME Version

The 386|VMM runtime version is supplied to allow redistribution of 386|VMM with a protected mode application. Distributed as a file named VMMDRVB.EXP, it must be "bound" to the protected mode application and to the runtime version of 386|DOS–Extender with BIND386. This binding process combines the runtime version of 386|DOS–Extender (RUN386B.EXE), the VMMDRVB.EXP file, and the application .EXP file into a single .EXE file which can be run by typing the file name from the MS-DOS command prompt. Please see the *BIND386 Utility Guide*. The following example binds a protected mode application BIGSORT.EXP with the RUN386B.EXE and VMMDRVB.EXP files to produce a file named BIGSORT.EXE:

```
bind386 run386b vmmdrvb bigsort
```

The bound program BIGSORT.EXE is then run by typing the file name:

```
bigsort
```

With bound application programs, the entire command line is passed to the protected mode application for processing. If it is necessary to use 386|DOS–Extender command line switches, they must be placed in an environment variable or configured into the bound .EXE file with the CFIG386 utility program. Please see the *386|DOS–Extender Reference Manual* for more information on command line switches, and the *CFIG386 Utility Guide* for details on configuring switches into programs.

2.3 Command Line Switches

Command line switches are used to change the default operation of 386 | DOS-Extender and 386 | VMM. This manual documents the 386 | DOS-Extender command line switches that apply specifically to operation with virtual memory. These switches can be used with 386 | DOS-Extender even if 386 | VMM is not used. Except where noted, the 386 | VMM-specific switches are ignored by 386 | DOS-Extender when used without virtual memory. By default, 386 | VMM:

- Places the swap file used for paging in the root directory of the device from which the application program was loaded. Please see Section 3.1 for a description of how 386|VMM uses the swap file.
- Always increases the swap file size when the program allocates additional virtual memory.
- Uses a least-frequently-used algorithm for selecting the page to be replaced (written to the swap file), when bringing another page into memory.
- Updates the virtual page aging information (used by the page replacement algorithm) every four seconds.

Command line switches begin with a minus sign (-) character, followed by the name of the switch (e.g., -LFU). There are two forms of each switch name: a long form and a short form. Any argument to the switch must immediately follow the switch name, with a space as a separator (e.g., -VSCAN 4000). If conflicting switches are given on a command line, the last (right-most) switch takes precedence.

Some of the command line switches take a number as an argument. By default, the number is considered to be a decimal (base 10) number. Hexadecimal numbers may be specified by appending the character "h" or "H" to the number. The following two examples both give the same number as an argument to the switch -VSCAN:

```
run386 -vscan 2048 -vmfile vmmdrv bigsort
run386 -vscan 800h -vmfile vmmdrv bigsort
```

2.3.1 Virtual Memory Driver Switches

The -VMFILE switch specifies the name and location of the development version of 386|VMM to be loaded by 386|DOS-Extender during initialization. Using this switch causes the application program to run in a virtual memory environment.

The -VMFILE switch is normally not used with bound applications (programs which have the redistribution versions of 386|DOS-Extender and 386|VMM bound to them in a single .EXE file). If the -VMFILE switch is used with a bound application, the virtual memory driver specified with the switch will be loaded, instead of the virtual memory driver that is bound to the application. This can be useful for testing a bound application with a later release of 386|VMM.

The -NOVM switch instructs 386|DOS-Extender not to load a virtual memory driver, regardless of whether the -VMFILE switch was used, or whether 386|VMM is bound to the application program. It causes the program to run in a non-virtual environment.

Syntax:

```
-VMFILE filename
-NOVM
```

Short form:

```
-VM filename
-NOVM
```

Example:

```
run386 -vmfile vmmdrv hello
run386 -vm \pharlap\vmmdrv.exp hello
cfig386 hello.exe -novm
```

2.3.2 Swap File Location and Name Switches

The -SWAPNAME switch specifies the swap file name. If this switch is not used, a DOS temporary (unique) file name is used. If the file name specified with the -SWAPNAME switch is already present on the swap device, 386|VMM refuses to run.

The -SWAPDEFDISK switch specifies that the current default disk drive be used for the swap file. If this switch is not used, the default device for the swap file is the disk on which the application program is located.

The -SWAPDIR switch specifies the device and directory in which to place the page swap file. The default location for the swap file is the root directory of the device from which the application program was loaded. This switch is useful for placing the swap file on a device which has sufficient free space to allow the swap file to grow as needed.

Relative path names are not permitted with the -SWAPDIR switch. The path name must start with a device name or the slash (/) or backslash (\) character (the root directory). If the path name does not start with a device, 386 | VMM prepends the name of the current default disk.

The directory name must not end with a backslash, because 386 | VMM appends a backslash before adding the name of the swap file.

Syntax:

```
-SWAPDIR dirname
-SWAPNAME filename
-SWAPDEFDISK
```

Short Form:

```
-SWD dirname
-SWN filename
-SWDEFD
```

Example:

```
run386 -swapdir d: -vm vmmdrv hello
run386 -swd e:\tmp -swn swapfile -vm vmmdrv hello
run386 -swdefd -vm vmmdrv f:\pgms\hello
```

2.3.3 Page Replacement Policy Switches

386 | VMM supports two switch selectable page replacement policies. The page replacement policy defines the algorithm used to select a page to be swapped to disk when a page already on disk needs to be brought into memory. The performance of a program in a virtual memory

environment depends to some extent on whether the system usually replaces pages that are not needed again for a long time; ideally, the page selected for replacement is the page not referenced by the program for the longest time into the future. Depending on the memory referencing patterns of an application, one of the page replacement algorithms supported by 386|VMM may yield better performance than the other.

The -LFU switch selects the Least Frequently Used replacement policy. A reference frequency count is kept with each page. Periodically, the page tables are scanned, and the count is either incremented or decremented, depending on whether the page was referenced since the last scan. The page with the lowest count (the least-frequently-used page) is the page selected for replacement. This is the default page replacement policy if no switches are used.

The -NUR switch selects the Not Used Recently replacement policy. This algorithm chooses a page for replacement based on whether the page has been accessed by the program, and whether it is dirty (its contents have been modified). Periodically, the page tables are scanned to mark all pages not accessed. The page accessed information thus identifies pages which have been referenced recently (since the last page table scan).

The -VSCAN switch selects how frequently the page tables are scanned in order to update the page aging information used by the page replacement policy. Changing the scan period affects which pages are selected for replacement, and therefore, affects program performance. The -VSCAN switch takes as an argument a time expressed in milliseconds (ms). The minimum value which may be given is 1000 ms (one second). The default scan period is 4000 ms. Note that 386|VMM assumes the timer tick interrupt occurs 18.2 times per second; application programs which change this standard timer operation must adjust the value specified with the -VSCAN switch appropriately.

The -VSLEN switch determines the maximum amount of linear address space to process on each page table scan. The default is FFFFFFFFh (four GB). While page table scans are very fast, for programs which use a very large virtual address space (32 MB or more) it may be desirable to limit the size of each page table scan. The minimum allowed value for this parameter is one megabyte.

The -VSCAN and -VSLEN parameters can also be changed at runtime with the Set 386|VMM Parameters system call (function 252Eh).

Page replacement algorithms are discussed in more detail in Section 4.4.

Syntax:

```
-LFU
-NUR
-VSCAN nmilliseconds
-VSLEN nbytes
```

Short Form:

```
-LFU
-NUR
-VS nmilliseconds
-VSL nbytes
```

Example:

```
run386 -lfu -vm vmmdrv hello
run386 -nur -vm vmmdrv hello
run386 -nur -vscan 2000 -vm vmmdrv hello
run386 -vslen 800000h -vm vmmdrv hello
```

2.3.4 Swap File Growing Policy Switches

The page swap file can potentially grow very large, if the virtual address space required by the program is large. Under these circumstances, it is possible to run out of disk space. The tradeoff is using up more disk space than is actually needed versus taking the risk of running out of swap space during a page fault, in which case 386|VMM is forced to abort the program. Section 4.3 discusses this topic, and the use of these switches, in more detail.

The -SWAPCHK switch is used to select when the size of the swap file is increased by 386|VMM. The -SWAPCHK MAX setting causes the swap file to grow whenever the virtual address space of the program is increased. The size of the swap file is always set to the size of the program's virtual address space, which is the largest size that could possibly be needed. If the swap file cannot be increased when a memory allocate system call is made, the memory allocate call returns failure, so

the program can deal with the condition gracefully. This is the safest setting, because it guarantees that 386|VMM will always have swap space available when a page fault occurs. It does, however, result in the largest swap file. Disk space problems can sometimes be solved by placing the swap file on a different disk drive with the -SWAPDIR switch.

The default setting is -SWAPCHK FORCE, which still causes the swap file to grow whenever additional virtual memory is allocated. However, the size to which it is increased is smaller than the virtual address space for the program, while still large enough to guarantee that sufficient swap space is available when a page fault occurs. This setting is a good compromise. It results in a smaller swap file, but ensures that no unexpected program aborts will occur. However, if this setting is used and too large a value is specified by the -CODESIZE switch, it can result in an out of swap space condition during a page fault (please see Section 4.3 for details).

The -SWAPCHK ON setting does not increase the swap file when virtual memory is allocated; instead, the swap file size is increased by the page fault handler as it needs new swap space. When additional virtual memory is allocated, the amount of free space on the disk is checked to make sure there is sufficient free space, using the same swap space requirements as those imposed by the -SWAPCHK FORCE setting. However, the swap file size is not actually increased until the space is needed. This setting minimizes the size of the swap file; but if the program uses up disk space for another purpose between the time the memory allocate is performed and additional swap file spaced is needed, a fatal out-of-swap-space error may occur in the page fault handler. In addition, there will be some performance degradation, because it is more expensive to increase the swap file one page at a time than to increase it in large chunks when additional virtual memory is allocated. For these reasons, it is normally better to use the -SWAPCHK FORCE setting.

The -SWAPCHK OFF setting disables all swap space checking when virtual memory is allocated. The swap file is increased as needed when a page fault occurs. As with -SWAPCHK ON, this minimizes swap file size, but leaves the program vulnerable to out-of-swap-space fatal errors when a page fault occurs. If this setting is used, the program should install an out-of-swap-space handler that attempts either to create more swap space,

or clean up and exit, when this condition occurs. Please see Appendix D.2 for more information.

Syntax:

```
-SWAPCHK OFF
-SWAPCHK ON
-SWAPCHK FORCE
-SWAPCHK MAX
```

Short Form:

```
-SWC OFF
-SWC ON
-SWC FORCE
-SWC MAX
```

Example:

```
run386 -swapchk force -vm vmmdrv hello
run386 -swc off -noswfg -vm vmmdrv hello
run386 -swc on -codes 9000h -vm vmmdrv hello
```

The -CODESIZE switch specifies the number of bytes of code which can be paged to disk without seriously affecting program performance. It is equal to the total size of the program's code, in bytes, minus the program's code "working set," that is, the amount of code that needs to be in memory at any given time to avoid excessive paging. This information is used with the -SWAPCHK FORCE and -SWAPCHK ON settings to calculate the minimum swap space required. Increasing the value reduces the swap file size. Please see Section 4.3 for more information. Specifying too large a value with this switch may result in unacceptable program performance, or even in fatal out-of-swap-space errors.

Syntax:

```
-CODESIZE nbytes
```

Short Form:

```
-CODES nbytes
```

Example:

```
run386 -swapc off -codes 20000h -vm vmmdrv hello
```

The -SWFGROW1ST and -NOSWFGROW1ST switches specify what the page fault handler should do when it needs a page in the swap file and one is not available. -SWFGROW1ST is the default, and causes the page fault handler, first, to attempt to make the swap file grow, and then if that fails, to attempt to take a swap file page away from a virtual page currently in memory. This can be done because a page in memory does not need space in the swap file. The -NOSWFGROW1ST setting reverses the order; it causes a swap page to be taken away from an in-memory page first, and the swap file to be increased only if no in-memory page owns a page in the swap file. The tradeoff is performance versus disk space. Disk space requirements are reduced if the -NOSWFGROW1ST switch is used, but program performance suffers. Please see Section 4.3 for an explanation of the tradeoffs involved.

Syntax:

```
-SWFGROW1ST
-NOSWFGROW1ST
```

Short Form:

```
-SWFG
-NOSWFG
```

Example:

```
run386 -noswfgrow -swapc on -vm vmmdrv hello
```

The -MINSWFSIZE switch forces creation of a minimum size swap file at program load time. It specifies a minimum size, in bytes, for the swap file created at startup. If this switch is not used, the initial swap file size is determined by the -SWAPCHK setting, the amount of physical memory available, and the load-time memory requirements of the program. If 386|VMM is unable to create a file of the specified size, it will refuse to run.

The -MAXSWFIZE switch limits the maximum disk space that is allocated to the swap file. It specifies a size, in bytes, beyond which the swap file is never increased. If this switch is not used, the only upper bound on swap file size is the amount of free space available on the disk.

Syntax:

```
-MINSWFSIZE nbytes
-MAXSWFSIZE nbytes
```

Short Form:

```
-MINS nbytes
-MAXS nbytes
```

Example:

```
run386 -maxs A00000h -vm vmmdrv hello
```

2.3.5 Flushing the Swap File to Disk Switches

The swap file is never closed during normal 386 | VMM operation. It is created when the program starts, and is deleted after the program terminates. This causes no problems during normal operation. But if your program has a bug that causes a reboot, or if you reboot your computer by hand while the application is running, the directory entry for the swap file never gets updated on the disk. The result is that you have to run the DOS CHKDSK program after a reboot to reclaim lost disk clusters which were allocated to the swap file.

The -FLUSHSWAP switch forces 386 | VMM to update the disk directory entry for the swap file each time the swap file size is increased. This is done by duplicating the file handle and closing the duplicate, or by closing the swap file and reopening it if no DOS file handles are available.

If you use the -FLUSHSWAP switch, you do not have to run CHKDSK to reclaim lost disk space after a reboot. However, you still have to delete the swap file with the DEL command to reclaim the disk space.

Because of the extra file I/O for the file close each time the swap file size is increased, the -FLUSHSWAP switch may incur a slight performance penalty. You may want to turn on this switch during program development, and turn it off when the product is shipped.

Syntax:

```
-FLUSHSWAP
```

Short Form:

```
-FLUSH
```

Example:

```
run38 -flush -vm vmmdrv hello
```

2.3.6 The Limit Program Virtual Memory Size Switch

By default, there is no limit on the amount of virtual memory that may be allocated by an application program. In practical terms, the virtual memory consumption of a program is limited by the amount of available disk space for the swap file.

The -MAXPGMMEM switch specifies the maximum number of bytes of virtual memory that the program can allocate. Attempts to allocate more than this amount result in the allocate memory system call returning an out-of-memory error.

Syntax:

```
-MAXPGMMEM nbytes
```

Short Form:

```
-MAXP nbytes
```

Example:

```
run386 -maxp A00000h -vm vmmdrv hello
```

2.3.7 Paging Out of the .EXP File Switch

The -DEMANDLOAD switch prevents the program from being initially loaded into memory; instead, it is paged in as needed. This reduces load times and guarantees that program pages are not read into memory until needed, but may actually increase overall program run times on machines

with a lot of physical memory, due to the overhead of reading the program into memory in 4K pages rather than all at once.

The -DEMANDLOAD switch is ignored if the program is a packed .EXP file or if 386|VMM is not used.

The -NOPGEXP switch forces 386|VMM not to page out of the .EXP file. When the -NOPGEXP switch is used, 386|DOS-Extender reads the entire program into memory at load time, and places all swapped out pages (including code pages) in the swap file. This can be useful if the application program is loaded from a network server and paging over the network is causing performance degradation.

Syntax:

```
-DEMANDLOAD
-NOPGEXP
```

Short Form:

```
-DEM
-NOPGE
```

Example:

```
run386 -dem -vm vmmdrv hello
run386 -nopgexp -vm vmmdrv hello
```

2.3.8 Locking the Program Stack

When the program is run at privilege level 0 (the -PRIVILEGED switch is used), the program stack must be locked in memory so no page faults can occur on stack references. By default, 386|VMM locks the entire initial program stack, which is identified in the .EXP file if the program was linked with version 3.0 or later of 386|LINK. If an earlier version of the linker is used, 386|VMM locks the first 8K of the program's initial stack.

The -LOCKSTACK switch overrides the default stack size calculation. It specifies a size, in bytes, to lock on the initial program stack.

The -LOCKSTACK switch is ignored if 386|VMM is not used, or if the program runs at privilege level 3 (the -UNPRIVILEGED switch is used).

Programs running at level 3 do not need a locked stack because page faults on stack references are permitted for level 3 programs.

Please see section 3.3 for more information.

Syntax:

```
-LOCKSTACK nbytes
```

Short Form:

```
-LOCKS nbytes
```

Example:

```
run386 -locks 4000h -vm vmmdrv hello
```

2.3.9 Creating a Page Fault Log File

The -PAGELOG switch enables writing 386|VMM paging information to a log file. It is recommended that -DEMANDLOAD be used when -PAGELOG is used, in order to obtain paging information for the program initialization phase.

The log file is a sequence of binary records. Data values are stored in the file in least-significant-byte first format. The first byte in each record is a record number. The records are defined as follows:

Record Number	Length (Bytes)	Field Offset	Data Type	Description
0	7			segment base changed
		1	WORD	segment selector
		3	DWORD	segment linear base address
1	7			segment limit changed
		1	WORD	segment selector
		3	DWORD	segment limit, in bytes
2	6 or 10			page fault occurred
		1	BYTE	flags byte
		2	DWORD	linear address of page fault
		6	DWORD	(optional, see flags description) linear address of replaced page
3	5			size of swap file increased
		1	DWORD	new swap file size, in bytes
4	variable			application record entry
		1	WORD	size of record, in bytes, including record number and size word
		3	variable	record data
5-255	variable			reserved for future expansion, will always have a size word following the record number, so software processing a page log file should always use the size word to skip over records 5 or greater.
		1	WORD	size of record, in bytes, including record number and size word.
		3	variable	record data

The flags byte for a page fault record (number 2) has the following bit definitions:

Bit mask	Meaning
01h	set if page read in from either the swap file or a data file, cleared if page zeroed
02h	set if page read in from the swap file, cleared if read in from a data file
04h	set if page has been in memory before and was swapped out, cleared if first time page was read into memory
08h	set if another virtual page had to be replaced and the record size is 10 bytes (the field at offset 6 containing the linear address of the replaced page is present), cleared if a free physical page was allocated and the record size is six bytes.
10h	set if the replaced virtual page had to be written to the swap file or a data file, cleared if the replaced page was discarded because it had not been modified.
E0h	reserved for future expansion, always cleared.

The term "data file" refers to a disk file, other than the swap file, used to hold virtual pages. This includes the .EXP file for the main program, the .EXP file for any program loaded with the Load Program (2529h) or Load for Debug (252Ah) system calls, and any data files mapped with system call 252Bh subfunction 3.

If the segment base and/or limit of aliased segments changes (such as segments 000Ch and 0014h, the program code and data segments), an entry is written to the page log file for each of the aliased segments.

Syntax:

```
-PAGELOG filename
```

Short Form:

```
-PL filename
```

Example:

```
run386 -pagelog hello.log -demandload -vm vmmdrv hello
```

Chapter 3

Programming in a Virtual Memory Environment

This chapter provides background information on the basics of programming using 386|VMM. The first section discusses the mechanics of how 386|VMM performs its functions; the second section covers the considerations of virtual address space and physical memory. The third section describes stack locking for privilege level 0 programs. The fourth section discusses unmapped pages and null pointers; the fifth provides a basic overview of mixed real- and protected-mode programs. The final sections explain the EXEC system call; program crashes; and interrupt handlers.

3.1 Fundamentals of Virtual Memory

Programs in a virtual memory environment can grow larger than the physical memory available, by keeping the portions of the program which will not fit in physical memory on disk. In a demand-paged virtual memory environment such as that provided by 386|VMM, programs are treated as an array of equal-sized memory blocks, called "pages." On the 80386, each page is four kilobytes in size.

Program pages are referred to as "virtual pages." The physical memory on the computer is also divided up into pages, called "physical pages." At any given point during the execution of a program, the program is actively accessing only a subset of its code and data. The virtual pages which contain the code and data being accessed are the virtual pages which must be kept in physical memory pages. Any virtual pages not being used can reside on disk until they are referenced by the program. As long as a program's references are reasonably localized (the number of virtual pages which are actively being accessed by the program is less than the number of available physical pages) at any given point in its execution, the

program will perform well in a virtual memory environment. Programs which frequently reference most of their address space (e.g., repeatedly making linear scans through a huge amount of memory) do not perform well under virtual memory, because they continuously require virtual pages to be brought into physical memory from disk, a condition known as "thrashing."

The job of 386|VMM is to maintain data structures known as page tables, to keep track of which virtual pages are in memory and which are on disk, and to bring virtual pages into memory as they are referenced by the program. This is implemented by using the hardware paging capabilities built into the 80386/80486 processor. When an attempt is made to reference a virtual page which is not in memory, the processor generates a page fault exception. 386|VMM then brings the page into memory and restarts the instruction which caused the page fault. To the application program, it appears as though the virtual page was always in memory.

When a page fault occurs and a virtual page is brought into memory, a physical page must be allocated for it. Typically, there are no free physical pages available, so a physical page must be obtained by taking one from another virtual page, which is first saved on disk. This operation is known as "replacing a page." Ideally, the virtual page chosen for replacement is the page that is not used again for the longest time into the future. The worst choice that could be made is a virtual page that will be referenced by the very next instruction in the program. Page replacement algorithms are discussed in Section 4.4.

Virtual pages on disk are kept in a file called the "swap file." A swap file is automatically created by 386|VMM when the program is loaded, and it is deleted when the program terminates. The device and directory on which the swap file is kept can be selected with the -SWAPDIR switch.

The physical pages available to the program are allocated either out of conventional memory (below one megabyte), or extended memory (above one megabyte). The reason for the distinction is that MS–DOS only knows about conventional memory; more details on this are given in the *386|DOS–Extender Reference Manual*. Normally, a program wants to have all the available physical memory for its own use, because then it can maximize the number of virtual pages kept in memory, thus optimizing program performance. However, there are occasions when it is desirable

to reduce the physical memory usage of a program in order to make physical memory available for use by another program; for example, before performing an EXEC to a child program. This is done by making system calls to change the program's conventional memory or extended memory usage with INT 21h functions 2521h, 2525h, and 2536h. Please see Appendix C.

3.2 Initial Program Memory Space

When a program is first loaded into memory, its virtual address space is as large as the amount of memory required to load the program, plus at least the amount of extra space specified with the link-time -MINDATA switch (default value zero). This is guaranteed under 386|VMM, even if there is less physical memory on the computer than the minimum virtual address space required. If the machine has more physical memory than the program's minimum virtual address space requirements, the initial address space is set to the smaller of either the amount of physical memory available, or the program's load requirements plus the extra space specified with the -MAXDATA switch (default value four gigabytes). It is usually best to link a virtual memory program such that it always has the same size initial address space, regardless of the amount of physical memory available on the machine. This is done by giving the -MAXDATA switch the same value as the -MINDATA switch. Use of the -MINDATA and -MAXDATA switches is also discussed in Section 2.1.

The size of the program's virtual address space can be modified at run time by using the memory allocation system calls (INT 21h functions 48h, 49h, 4Ah, 2529h, and 252Bh). If an attempt to allocate more memory returns an insufficient memory error (error code 8), additional information about the cause of the error can be obtained from system call 2519h. The most likely reason for memory allocation to fail under 386|VMM is running out of swap space on the disk.

3.3 Program Stack

Since hardware interrupt handlers must not cause page faults, they must be invoked with a stack that is not in virtual memory (please see section 3.8). If the program is running at privilege level 3 (-UNPRIV), this is

automatic. The application program stack can be virtual memory, and because an interrupt will always cause a privilege ring transition, a locked stack will always be created for the interrupt handler.

If the program is running at privilege level 0 (-PRIV), no new stack will be created when interrupts occur. Because of this, an application program running at level 0 **must** keep its stack locked at **all** times. By default, 386 | DOS-Extender will lock the program's entire internal stack at load time. The amount of stack memory to lock can be changed from the default value of "all" by using the -LOCKSTACK switch.

If a privilege level 0 application ever switches stacks, it **must** lock the new stack before switching to it.

3.4 Unmapped Pages and Null Pointers

It is possible for an application program to have unmapped pages within its virtual address space. Unmapped pages are virtual pages which are neither in physical memory nor on disk. Unmapped pages can be created at link time by using the -OFFSET switch to create unmapped pages at the beginning of the program's virtual address space, or at run time by using the Memory Region Page Management (252Bh) or Add Unmapped Pages (252Ch) system calls.

When the page fault handler gets a page fault on an unmapped page, it is, of course, unable to bring the page into memory because it is not on the disk. Instead, it transfers control to an alternate page fault handler. The default alternate page fault handler under 386 | DOS-Extender terminates the program and prints out a message giving the program location of the instruction causing the page fault. The default alternate page fault handler under 386 | DEBUG or 386 | SRCBug gives you control at the debug prompt and lets you examine code and data as usual to determine the cause of the error.

The most common method of creating unmapped pages is using the -OFFSET switch at link time. The -OFFSET switch creates unmapped pages at the beginning of the program's virtual address space; that is, at offset zero in segments 000Ch and 0014h (the program's code and data segments). This is useful for detecting null pointer references in the

application program. A common programming error is to use a pointer variable to reference memory before initializing the pointer. In many cases, the value that happens to be in the pointer variable will be zero or some other small number; thus, an alternate page fault will occur when the pointer is used.

It is possible for application programs to install their own alternate page fault handler, instead of the default handler installed by 386 | DOS-Extender or the debuggers. Please see Appendix D.3 for details.

3.5 Mixed Real and Protected Mode Programs

Some applications have a certain amount of code which executes in the real mode of the 80386; such programs are known as mixed mode programs. Since page faults cannot occur when the processor is executing in real mode, all real mode code and data must be in virtual memory which can never be paged out to disk.

There are a variety of techniques for mixing real and protected mode code in a single program; please see the *386 | DOS–Extender Reference Manual* for details. The technique of particular interest under 386 | VMM is combining the real mode code and the protected mode code in a single .EXP file. Techniques which separate the real mode and protected mode code into two separate program links do not present any special problems under virtual memory.

Mixed-mode programs linked in a single .EXP file must make sure the real mode code and data end up in conventional memory. The recommended method is to use the segment ordering rules of the linker to force the real mode code and data to the beginning of the program, and then to use the -REALBREAK switch to guarantee the real mode code and data are placed in conventional memory when the program is loaded. Again, please see the *386 | DOS–Extender Reference Manual* for details. When this method is used under 386 | VMM, the program loader also locks all -REALBREAK pages in memory so they cannot be swapped to disk; the application program merely needs to be careful never to unlock those pages with the Unlock Pages system call. If the program uses some method other than -REALBREAK to force its real mode code and data into conventional memory, it is also responsible for explicitly locking those

pages with the Lock Pages in Memory system call (INT 21h function 251Ah, or function 252Bh subfunction 5) so they cannot be swapped.

Conventional memory which is allocated by calling MS–DOS directly either from real mode, or by using system calls 25C0h or 25C2h in protected mode, is not virtual memory and will never be paged to disk. Therefore, it can be used safely by real mode as well as protected mode code. Memory allocated in protected mode via system calls 48h or 4Ah is virtual memory and can always be paged to disk unless explicitly locked in memory. Even when it is in memory, only chance dictates whether the virtual memory page is located in a physical conventional memory page or extended memory page. Thus, memory allocated with system calls 48h or 4Ah must never be accessed by real mode code.

3.6 The EXEC System Call

For an application program to successfully EXEC to a child program, there must be sufficient physical memory free to load the child into memory. If the child program is a standard MS-DOS real mode program, there must be enough conventional memory available for its requirements. If the child is a protected mode program, there must be sufficient conventional memory available to load a second copy of 386 | DOS–Extender (approximately 70 kilobytes) and enough additional memory (either conventional or extended) available to load the child program itself. Of course, if the child program also runs with virtual memory, its needs for physical memory can be quite modest.

By default, all available physical memory is allocated to the application program. This is done because the more physical memory there is, the more program virtual pages can be in physical memory, resulting in less paging and better performance. Thus, before performing an EXEC to a child program, some physical memory must be freed so there will be enough memory for the child to run. System calls 2521h, 2525h, and 2536h are used to adjust physical extended memory usage and physical conventional memory usage. Before performing an EXEC to a child, these calls are used to free up some physical memory for the child to run in. After the child program completes its processing, these calls are used to take the physical memory back for use again by the parent program, so that its performance does not suffer.

For parent programs which make a lot of EXEC calls and which do not need large amounts of physical memory to run efficiently, it may be more desirable to always leave physical memory free, rather than shrinking and expanding its memory usage around each EXEC call. This is done preferably by freeing up memory during initialization with the 2521h, 2525h, or 2536h system calls, or it can be done with command line switches when the program is run. The -MINREAL and -MAXREAL switches are used to leave conventional memory free when the program is started up. The -MAXEXTMEM, -MAXVCPIMEM, and -MAXXMSMEM switches are used to limit consumption of direct extended memory, VCPI memory, and XMS memory, respectively. More information on the use of these switches, and on performing an EXEC to a child program, is contained in the *386|DOS-Extender Reference Manual.*

The INT 21h function 4Bh EXEC call flushes the swap file to disk before performing the EXEC. This is done so the DOS CHKDSK program will not see unlinked clusters from the swap file. The INT 21h function 25C3h EXEC call does not flush the swap file. There is no good reason to use the 25C3h form of the exec call; it exists only for historical reasons.

3.7 Program Crashes

When a program crash causes a system reboot or similar failure, the page swap file never gets closed and deleted. This results in a loss of space on the disk. The DOS CHKDSK utility should be run after a program crash to reclaim the lost disk space. The swap file itself will be an empty file with a DOS temporary file name (which always begins with a decimal digit and looks like a random sequence of digits and letters); this can be deleted with the DOS DEL command.

If the -FLUSHSWAP switch is used, the swap file directory entry is updated each time the size of the swap file increases. With -FLUSHSWAP, you don't need to run CHKDSK to reclaim disk space, you only need delete the swap file.

3.8 Interrupt Handlers

Handlers for asynchronous events, such as hardware interrupts, must be written so that a page fault never occurs while the handler is executing. This is done by using the Lock Pages in Memory system call (251Ah, or 252Bh subfunction 5) to lock in memory all code and data pages, including the stack, which the handler references. The reason for this requirement is that MS–DOS is not reentrant, and the page fault handler makes calls to perform disk I/O to the swap file.

The above requirement applies to all protected mode code which can get control via an asynchronous event. For example, a mixed real and protected mode program might install all of its hardware interrupt handlers in real mode. However, if a handler makes a cross-mode call to a protected mode routine, that routine's code and data must be locked down so a page fault cannot occur while it is executing.

When an interrupt handler gets control, it always has a locked stack. If it switches to another stack, it must be sure to lock the new stack before switching to it.

Note that normally it is not necessary to lock handlers for software interrupts in memory, because software interrupts do not occur asynchronously.

Please see the *386 | DOS-Extender Reference Manual* for more information on writing interrupt handlers.

3.8.1 CTRL-C Interrupt Handler

The CTRL–C interrupt is always generated in real mode by MS–DOS when it sees a CTRL–C character in the keyboard buffer. A CTRL–C handler's code and data do not have to be locked in memory, even if the handler executes in protected mode. This is because MS–DOS makes sure it can be reentered before issuing a CTRL–C interrupt, and 386 | VMM solves the page fault handler reentrancy problem by disabling CTRL–C interrupts while disk I/O for a page fault is in progress.

3.8.2 Critical Error Interrupt Handler

The critical error handler is generated in real mode by MS–DOS for certain classes of errors, such as disk errors. If the program sets up a protected mode handler for this interrupt (with system call 2506h), the handler's code and data must be locked down, as is required for hardware interrupts.

Chapter 4

Performance of Virtual Memory Programs

This chapter discusses options which control the performance of 386|VMM. Most of the options are switch-selectable; for further information on the switches, please see Section 2.3. The options available are each discussed in a separate section: statistics checking; packed or unpacked .EXP files; increasing swap file size and how 386|VMM handles insufficient space in the swap file; page replacement algorithms; using a disk cache; locking pages in memory; and freeing physical memory.

4.1 Statistics Checking

The Get Memory Statistics system call (function 2520h) returns statistics on a program's memory usage and paging activity. Using this system call, an application program can include instrumentation to report statistics on its performance under 386|VMM. The program can then be run in various ways, using some of the options discussed in this chapter, to determine which combination of options yields the best performance.

The -PAGELOG switch causes 386|VMM to log paging activity in a disk file. You can use this capability to analyze the paging activity of your program. The 386|VMM distribution disk contains a simple program that reads the contents of a page log file and writes formatted output to the display.

4.2 Packed vs. Unpacked EXP Files

Packed protected mode programs are built by using the -PACK switch at link time. This option reduces the size of the .EXP file on the disk. However, it is generally not desirable to use this option when running a program under 386|VMM.

When 386|VMM is used with an unpacked program, it writes program pages to the swap file only after they are modified. All unmodified pages are simply discarded and read from the .EXP file the next time they are needed. Thus, code pages, which typically never are modified, never end up in the swap file. This reduces the size required for the swap file (depending on the swap file options selected), and also results in better performance when the program is first running, because no code pages need to be written to the swap file. Please see section 4.3.

This optimization cannot be used with packed .EXP files. Since the space savings in the swap file are typically greater than the space savings achieved by packing the .EXP file, it is usually better to use an unpacked .EXP file if both it and the swap file will be on the same disk (which is the default condition, unless the -SWAPDIR switch is used).

A second disadvantage of packed files is the amount of time required to load a program. The entire contents of a packed file must be read into the virtual memory at program load time, so that it can be unpacked. If the program size is much larger than the physical memory, this will result in thrashing and greatly increased initial load times.

4.3 Swap File Options

Frequently, the biggest problem associated with running a virtual memory program is the disk space requirements for the page swap file. It is, therefore, desirable to make the swap file as small as possible. However, it is also important to make sure 386|VMM never runs out of swap space while processing a page fault; if this happens, 386|VMM is forced to abort the program with an out-of-swap-space error.

Two switch-selectable options for dealing with this situation are available under 386|VMM:

- choosing when the size of the swap file is increased — either each time the program's virtual address space is increased, or only when page faults occur

- specifying the first place the page fault handler attempts to get a swap page when it runs out of space in the swap file — either increasing the swap file size by a page, or taking a swap page away from a virtual page that is currently in physical memory

The tradeoffs are larger disk space requirements versus poorer program performance. The sections below discuss the switches used to control options, and the implications of the available choices.

4.3.1 Minimum Swap File Size

The -SWAPCHK and -CODESIZE switches specify the minimum size of the swap file (if any) when the virtual address space used by the program is increased. When a system call is made to increase the program address space, an out-of-memory error is returned if the swap file size cannot be increased to satisfy this minimum size requirement. System call 2519h can be used to obtain more detailed information on the cause of an out-of-memory error, including whether it is caused by running out of swap space. The application program can then take appropriate action: for example, by adjusting its memory needs downward, or posting a message informing the user that more disk space is required in order to run the program. If, on the other hand, an option is selected that does not require the swap file size to be increased when additional virtual memory is allocated, the memory allocation call will almost always succeed, except for attempts to allocate so much virtual memory that there is not enough physical memory available for the additional system pages (e.g., page tables) required by 386|VMM. However, at some later point in the program execution, the page fault handler may run out of swap space on the disk and be forced to abort the application program with an error message.

The most conservative swap file size option is selected with the -SWAPCHK MAX switch setting. This setting makes the swap file as large as the virtual address space used by the program, which guarantees the program will never run out of swap space. However, this can result in a very large swap file, and typically some portion of the allocated space goes unused.

The default swap file option is -SWAPCHK FORCE, which also increases the swap file whenever additional virtual memory is allocated; however, the swap file size is only increased to the size of the program's virtual memory space minus the size of physical memory available. This results in a smaller swap file, but still guarantees there will always be sufficient space in the swap file, because pages in physical memory do not need to be in the swap file. Note that the swap file size may, under some circumstances, later be increased above this minimum size by the page fault handler if there is disk space available (please see section 4.3.2.) However, if there is no disk space, the page fault handler is still able to get a swap page by taking one away from a virtual page currently in physical memory.

The -CODESIZE switch can be used to reduce the minimum swap file size still further when it is used in conjunction with -SWAPCHK FORCE. The -CODESIZE switch specifies the size of the application program's code which does *not* need to be kept in memory in order to obtain good performance. That is, it is equal to the total code size for the program, minus the code working set size, or the amount of code which needs to be in memory at any given time to avoid excessive paging. Typically, this working set size is determined empirically; a good starting point is to assume the working set is approximately 40% of total code size. Therefore, the value specified with the -CODESIZE switch is approximately 60% of the total code size (which can be calculated from the information in the linker map file). 386|VMM reduces the minimum swap file size by the value specified with the -CODESIZE switch, so that the minimum swap size is the virtual address space size, minus the available physical memory, minus the -CODESIZE value.

The rationale behind the -CODESIZE value is that code pages do not get modified, so they can always be brought into memory from the .EXP file and do not need to be written to the swap file. However, for good performance, it is necessary for some minimum amount of code pages to be in physical memory; that physical memory is then not available for data pages, which must be kept in the swap file. This is the reason the switch value should take into account the size of code which needs to be in physical memory. If too large a value is used, thrashing may result in low disk space situations, because not enough code is in physical memory. In the worst case, using too large a value can result in an out-of-swap space error and a program abort in the page fault handler. This may

happen if the value used is larger than the program's total code size, or if the program modifies many of its code pages so that they have to be written to the swap file instead of brought in from the .EXP file. The value specified with the -CODESIZE switch is ignored if the program is linked as a packed .EXP file, because, in that case, code pages cannot be paged out of the .EXP file at run time, and instead must be written to the swap file in the same way as data pages.

When the -SWAPCHK ON switch setting is used, the swap file size is never increased when additional virtual memory is allocated. However, the amount of free disk space is checked when virtual memory is allocated. If it is not sufficient to satisfy the same minimum swap file size requirements that are imposed with the -SWAPCHK FORCE setting, an out-of-swap-space error is returned from the memory allocation system call. Thus, if the program is not allocating disk space, this setting ensures that the page fault handler has sufficient swap space; but the swap file is always increased a page at a time when page faults occur, instead of in chunks when virtual memory is allocated. Also, it is possible to get a fatal out-of-swap-space error, if the application program uses too much disk space for its own purposes between the time the memory allocation call is made and 386|VMM checks free disk space, and the time additional swap space is required by the page fault handler.

When the -SWAPCHK OFF switch setting is used, no checking whatsoever is performed when virtual memory is allocated, and an out-of-swap-space error is never returned from an allocation call. The swap file is increased as needed when page faults occur, in the same way as the -SWAPCHK ON setting. Of course, since no checking is performed, it is likely that a fatal out-of-swap-space error will occur in low disk space situations with no warning.

It is possible for the application program to use system call 2523h to install an out-of-swap-space handler. The page fault handler calls the out-of-swap-space handler when it runs out of swap space and cannot increase the size of the swap file. There is no need to install such a handler unless the -SWAPCHK ON or -SWAPCHK OFF settings are used. If one of these settings is used, it is advisable to write such a handler, so that the program can do something reasonably user-friendly if it runs out of swap space. Appendix D.2 describes how to write and install an out-of-swap-space handler.

4.3.2 When 386|VMM Increases the Swap File Size

When the page fault handler needs to swap a virtual page to disk and there are no free pages in the swap file, it has two options. It can make the swap file grow by a page, or it can reclaim a swap file page (search for a virtual page, currently in physical memory, which owns a page in the swap file, and take the swap page away from it). The page fault handler will always attempt both of these options, if necessary, but the order in which it attempts them is switch-selectable.

The -SWFGROW1ST switch is the default setting and causes the page fault handler to make the swap file grow first, and only reclaim a swap page if it is unable to increase the swap file size. This setting typically results in a larger swap file, but often yields better program performance.

The -NOSWFGROW1ST switch causes the page fault handler to reclaim a swap page first, and only make the swap file grow if there are no in-memory virtual pages which own pages in the swap file. This setting uses less disk space for the swap file, but program performance suffers.

An example of how reclaiming swap pages can adversely affect performance is the case where a swap page is taken away from an in-memory virtual page and that virtual page is later selected to be swapped to disk. If the page is marked dirty, there is no performance penalty, because it has to be written to the swap file anyway. However, if the page has not been modified, then an extra disk write must be performed. Because it is not already in the swap file, it must be written out; but if its swap page had not previously been reclaimed, the disk write would not have been necessary.

Whether or not program performance actually suffers when swap pages are reclaimed depends to some extent on the application program's data accessing behavior. For a virtual page in memory to own a page in the swap file, it must previously have been swapped to disk. This means it must have been modified; that is, it must be data (unless the program is linked as a packed .EXP file, in which case code pages also end up in the swap file; for this reason, performance of packed programs **definitely** suffers if the -NOSWFGROW1ST switch is used). Since the virtual page is

in memory, it must have been referenced again since it was written to disk, because that is the only time virtual pages are brought back into memory in a demand-paged system. Usually, data is likely to be modified when it is referenced. If this is the case, performance is no worse when swap pages are reclaimed. However, if, for example, the application program initializes large data tables and then repeatedly references the tables without modifying them, performance will suffer if the -NOSWFGROW1ST switch is used.

4.3.3 Shrinking the Swap File

Under normal operation, 386|VMM increases the size of the swap file when needed, but never decreases the size of the swap file. The entire swap file is deleted when the program terminates.

The Shrink Swap File system call (function 253Ch) can be used to reduce the size of the swap file while the program is running. Usually, it is not possible to shrink the swap file. However, if the application program frees large amounts of memory, the swap file becomes sparsely populated and it is possible to coalesce the swap file and then truncate it to a smaller size.

For pages to be freed in the swap file (which makes it possible to shrink the swap file), virtual memory pages must actually be freed. This is **not** done when you call the memory allocator in a compiler runtime library. To free virtual memory, it is necessary to free a segment (fuction 49h), reduce the size of a segment (function 4Ah), or free up allocated pages within a segment (function 252Bh subfunction 0).

You will normally have to write your own memory allocator or modify the allocator provided with the compiler runtime library before you can dynamically shrink the swap file.

4.4 Page Replacement Algorithms

When a page fault occurs and a virtual page needs to be brought into physical memory, a physical page usually must be obtained by writing some other virtual page to disk, or by replacing the page. The algorithm used to select the page to be replaced can have a significant effect on overall program performance. Ideally, the page selected is the one not

used again for the longest time into the future. In the worst case, the page selected is the next virtual page to be referenced.

Most virtual memory systems use a variation of a Least Recently Used (LRU) page replacement scheme, under the heuristic that pages referenced most recently are most likely to be referenced again in the near future. Implementing full LRU is prohibitively expensive, since it requires time stamping a page or manipulating a linked data structure for each page reference. Therefore, some approximation to LRU typically is selected. 386|VMM provides two such approximations. It is advisable to experiment with both of them, since the performance obtained by a specific page replacement algorithm depends to some extent on the memory referencing characteristics of the application program. Thus, one algorithm could yield optimum performance for program A, while program B might perform best with a different algorithm.

The page replacement algorithm used by default is Least Frequently Used (LFU) page replacement. This algorithm keeps a count of how frequently each page has been referenced. Periodically, (the time period is controlled with the -VSCAN switch) the page tables are scanned to update the LFU counts, depending on whether the page has been referenced since the last scan. When selecting a page for replacement, the page with the lowest LFU count is chosen.

The -NUR switch is used to select the Not Used Recently (NUR) page replacement algorithm. This algorithm is implemented by using the Accessed and Dirty bits maintained for each page by the 80386 processor. These two bits place every allocated page in one of four categories:

Category	Accessed	Dirty
1	0	0
2	0	1
3	1	0
4	1	1

Pages in the lower number categories are the most attractive to be selected for replacement. The longer the application program runs, the more likely it is that most pages will have the Accessed bit set. This is undesirable, since 386|VMM loses its ability to detect which pages have been recently referenced. Therefore, periodically (the period being

specified with the -VSCAN switch), the Accessed bits for all pages are cleared. Of course, this makes even active pages briefly vulnerable to replacement; but not for long, since active pages will be quickly referenced again. This resetting of the Accessed bit is how pages end up in category 2, which otherwise would not make sense.

4.5 Using a Disk Cache

Most computer manufacturers distribute a disk cache program with their machines. A disk cache program keeps recently referenced disk sectors in memory, so that the system does not need to get them from the disk the next time they are referenced. Since 386|VMM keeps a page swap file on disk, there is a considerable amount of disk I/O for paging when running programs much larger than the available physical memory. With heavy disk activity, the DOS file system needs to access the disk File Allocation Table (FAT) frequently. Since the FAT tends to stay in the disk cache, the overall program performance can often benefit from installing a small disk cache. It is not advisable to allocate a lot of memory to the disk cache, since this reduces the amount of physical memory available to 386|VMM, resulting in increased paging.

4.6 Locking Pages in Memory

Locking virtual pages in physical memory (via system call 251Ah) has a negative effect on program performance. The more pages are locked, the fewer physical pages are available for use by the rest of the program, resulting in increased paging activity. You should only lock pages which must always be in memory for some reason (e.g., because they are referenced by a hardware interrupt handler).

4.7 Freeing Physical Memory Pages

The Free Physical Memory Pages system call (function 251Ch) can be used to improve performance for some types of application programs. It is used to inform 386|VMM that a range of virtual pages will not be needed for a while. 386|VMM can then free up the physical memory pages allocated to those virtual pages, making them available for use by other

parts of the program. This call essentially augments the page replacement algorithm. Because the application program has specific knowledge about its own memory referencing behavior, it can use this call to assist 386|VMM in making good choices about page replacement.

This system call is appropriate for programs which manage large amounts of data in virtual memory. Frequently, the program manipulates the data for a period of time, and then either discards the data or performs some other task which does not reference that data for a subsequent period of time. Under such circumstances, program performance can frequently be improved by judicious use of this call.

386|VMM Fatal Error Messages

A small number of fatal errors can occur during a page fault. When one of these errors occurs, 386|VMM prints a fatal error message on the display, deletes the swap file, and terminates program execution. These error messages always begin with the identifier "386|VMM:". The messages listed below can be caused by errors in the application program. Any other fatal errors in the page fault handler are the result of an internal error in 386|VMM, in which case you should attempt to make the error reproducible and call Phar Lap Software at (617) 661-1510 for technical support.

```
Phar Lap fatal err 10030: 386|VMM: Couldn't reopen swap file after flushing it
```

If the -FLUSHSWAP switch is used, 386|VMM closes the swap file to flush it each time the size of the swap file is increased. Independent of -FLUSHSWAP, the swap file is flushed when an INT21h function 4Bh EXEC call is made (but not when an INT 21h function 25C3h EXEC call is made). This error occurs when 386|VMM closes the main swap file handle (because no duplicate handles are available), and then gets an error on the DOS call to reopen the swap file. The usual reason for the error is that DOS has run out of file handles. You should do one of the following:

- reduce the number of open files
- stop using the -FLUSHSWAP switch
- use the 25C3h form of EXEC rather than 4Bh.

```
Phar Lap fatal err 10031: 386|VMM: Page fault handler was reentered
```

Your program has caused a page fault inside a hardware interrupt handler or a DOS critical error handler. Make sure all code, data, and stack space you reference within these handlers is locked.

```
Phar Lap fatal err 10032: 386|VMM: Bad page table entry in pg flt hndlr:
1st err val = linadr, 2nd = pg tbl entry, 3rd = pg tbl info
```

The page fault handler has detected an invalid page table entry. If you install page replacement handlers as described in section D.3, check your handlers carefully for errors. Otherwise, this is probably an internal 386|VMM error; write down the error printout, try to make the error reproducible, and call Phar Lap technical support at (617) 661-1510.

```
Phar Lap fatal err 10033: 386|VMM: Page replacement routine selected an invalid
page: 1st err val = linadr, 2nd = pg tbl entry, 3rd = pg tbl info
```

If you install page replacement handlers as described in section D.3, check your handlers carefully for errors. Otherwise, this is probably an internal 386|VMM error; write down the error printout, try to make the error reproducible, and call Phar Lap technical support at (617) 661-1510.

```
Phar Lap fatal err 10034: 386|VMM: Error reading EXP file, or data file mapped in
virtual memory
```

A DOS error occurred when trying to read a page in from the program's .EXP file, or from a data file mapped with function 252Bh subfunction 3. If this error occurs right after returning from an EXEC to the DOS command shell, check to see if the user deleted the file. Otherwise, your program has probably corrupted part of DOS or 386|DOS-Extender.

```
Phar Lap fatal err 10035: 386|VMM: Error reading swap file
```

A DOS error occurred when trying to read a page in from the page swap file, or from a data file mapped with function 252Bh subfunction 3. If this error occurs right after returning from an EXEC to the DOS command shell, check to see if the user deleted the file. Otherwise, your program has probably corrupted part of DOS or 386|DOS-Extender.

```
Phar Lap fatal err 10036: 386|VMM: Page table entry not present in pg fault hndlr
```

A page selected for replacement (swapping to disk) is not in memory. If you install page replacement handlers as described in section D.3, check your handlers carefully for errors. Otherwise, this is probably an internal 386|VMM error; write down the error printout, try to make the error reproducible, and call Phar Lap technical support at (617) 661-1510.

```
Phar Lap fatal err 10037: 386|VMM: Error writing swap file
```

A DOS error occurred when trying to write a page into the page swap file, or to a data file mapped with function 252Bh subfunction 3. If this error occurs right after returning from an EXEC to the DOS command shell, check to see if the user deleted the file. Otherwise, your program has probably corrupted part of DOS or 386 | DOS-Extender.

```
Phar Lap fatal err 10038: 386|VMM: Out of space for swap file
```

The disk on which the page swap file is located is full. This error should only occur if -SWAPCHK ON or -SWAPCHK OFF is used, or if you specified too large a value with -CODESIZE. Either use -SWAPCHK FORCE, or write an out-of-swap-space handler (see section D.2) to handle this error.

Appendix B

Example Code on the Distribution Disk

Some virtual memory code examples are included on the distribution disk for the development version of 386|VMM. This appendix gives a brief description of the contents of the files.

VM.C, VMBLD.BAT — A simple C program that allocates a large data buffer and scans it to cause paging to occur. The VMBLD.BAT batch file builds it with the MetaWare High C compiler.

READLOG.C, READLOG.BAT — A simple C program that reads a paging activity log file created with the -PAGELOG switch and writes it to standard output in readable format. The READLOG.BAT batch file builds it with the Metaware High C compiler.

PFHNDLR.ASM — An example of an installable page replacement handler (please see Section D.3).

PHARLAP.H — A C include file containing definitions of constants and data structures for MS-DOS and 386|DOS-Extender system calls.

HW386.H, HW386.AH — C and assembly language include files containing definitions of constants and data structures for the 80386/80486 chip architecture.

DOSX.AH — An assembly language include file containing definitions of constants and data structures for 386|DOS-Extender system calls.

Appendix C

System Calls

This appendix specifies the 386 | DOS–Extender system calls which are directly related to virtual memory support. Except where noted, all of these system calls can be made even if 386 | VMM is not present. Some of them are useful even without virtual memory; the others either do nothing or perform their usual task innocuously if virtual memory is not present.

See the *386 | DOS–Extender Reference Manual* for a complete list of system calls. This appendix lists only those calls which directly relate to operation under 386 | VMM.

386 | DOS–Extender system calls are made using software interrupt 21h, with a value of 25h in the AH register, and a function code in the AL register. These system calls may only be made when executing in protected mode.

All registers, except those used to return results, are preserved across 386 | DOS–Extender system calls. All 386 | DOS–Extender system calls return with the processor carry flag set if an error occurred, and with the carry flag clear if the call succeeded. All illegal function values return with the carry flag set, EAX set to A5A5A5A5h, and all other registers unchanged.

VIRTUAL MEMORY SYSTEM CALLS SUMMARY

Function Number	Function Name
2519h	Get Additional Memory Error Information
251Ah	Lock Pages in Memory
251Bh	Unlock Pages
251Ch	Free Physical Memory Pages
2520h	Get Memory Statistics
2521h	Limit Program's Extended Memory Usage
2522h	Specify Alternate Page Fault Handler
2525h	Limit Program's Conventional Memory Usage
252Bh, BH=3	Map Data File Into Allocated Pages
252Bh, BH=4	Get Page Types
252Bh, BH=5	Lock Pages
252Bh, BH=6	Unlock Pages
252Bh, BH=7	Free Physical Pages, Retaining Data
252Bh, BH=8	Free Physical Pages, Discarding Data
252Dh	Close VMM File Handle
252Eh	Get/Set 386|VMM Parameters
252Fh	Write Record to 386|VMM Page Log File
2536h	Minimize/Maximize Extended/Conventional Memory Usage
253Ch	Reduce Size of Swap File

INT 21h **Get Additional Memory Error Information**
AX = 2519h

INPUTS: None

OUTPUTS: Carry flag = clear
EAX = error code
= 0, if no error
= 1, if out of physical memory
= 2, if out of swap space (unable to grow swap file)
= 3, if out of LDT entries and unable to grow LDT
= 4, if unable to change direct extended memory allocation mark
= 5, if maximum program virtual size (selected with -MAXPGMMEM) exceeded
= FFFFFFFFh, if paging is disabled (-NOPAGE switch used)

This call provides additional information when a memory allocator system call returns an out-of-memory error (error number 8). The error code returned applies to the last system call made which returned an out-of-memory error.

The most likely error code returned when running under 386|VMM is error 2 (out of space on the disk for the swap file). The best way to deal with this condition is to free up some space on the disk. The swap file can also be placed on a different disk with more free space using the -SWAPDIR switch, or a different swap file increasing policy can be selected (please see Section 4.3).

When running without 386|VMM, the most likely error condition is error 1 (out of physical memory). If this error occurs under 386|VMM, it means the memory allocation request was large enough that 386|VMM needs to allocate system pages (such as page tables), and there is not enough physical memory available for the required system pages. The probable reason for this is that the application has locked too many pages in memory. Another possibility is an unreasonably large allocation

request (e.g., four gigabytes) has been made that requires many system pages to be allocated on a machine without much physical memory.

Error 3 means either that the LDT is already its maximum size (64 kilobytes, or 8K segments), or that there is no physical memory available to increase the LDT size.

Error 4 means that another program (such as a terminate-and-stay-resident program) is attempting to allocate physical extended memory independently. If 386 | DOS-Extender detects this condition, it abandons any attempt to allocate more direct extended memory for fear of conflicts.

Error 5 means that the program attempted to allocate more virtual memory than the limit specified by the -MAXPGMMEM switch.

INT 21h
AX = 251Ah **Lock Pages in Memory**

INPUTS:
EDX = the number of pages to lock
If BL = 0:
ECX = linear address of first page to lock
If BL = 1:
ES:ECX = pointer to first page to lock

OUTPUTS:
If success:
Carry flag = clear
If failure:
Carry flag = set
EAX = error code
= 8, if memory error
= 9, if invalid address range

This call locks the specified address range in memory, paging in any virtual pages which are swapped to disk. Please see also function 252Bh subfunction 5, which performs the same operation and may be easier to use.

The start address is not required to fall on a page (4K) boundary; the first virtual page locked is the page in which the start address is located. The entire range of pages to be locked is required to be within the address space allocated to the application program. It is legal for pages in the specified range to be already locked, or to be unmapped pages or mapped physical device pages. If an unmapped or mapped page table entry is encountered, the page table entry is not modified.

The most common reason for using this call is to lock code and data needed by interrupt handlers (please see Section 3.8) or virtual memory handlers (please see Appendices D.2 and D.3) in memory, because such handlers are not permitted to cause page faults.

Pages are also sometimes locked in memory when it is known they will be used heavily. Making sure they never get swapped to disk sometimes improves performance. However, pages which are referenced frequently

are unlikely to get swapped to disk in any case, because 386|VMM always attempts to choose pages for replacement which are not heavily used. Locking too many pages in memory can actually degrade system performance, so use of the Lock Pages in Memory system call to tune performance is not recommended.

INT 21h **Unlock Pages**
AX = 251Bh

INPUTS: EDX = the number of pages to unlock
If BL = 0:
ECX = linear address of first page to unlock
If BL = 1:
ES:ECX = pointer to first page to unlock

OUTPUTS: If success:
Carry flag = clear
If failure:
Carry flag = set
EAX = error code
= 9, if invalid address range

This call unlocks any locked pages in the specified address range. See also function 252Bh subfunction 6, which performs the same operation and may be easier to use.

The start address is not required to fall on a page (4K) boundary; the first page unlocked is the page in which the start address is located. The entire address range is required to be within the address space allocated to the program. It is legal for pages in the specified range to be already unlocked, or to be unmapped pages or physical device pages. Note that mixed real and protected mode applications should be careful not to unlock pages which contain the real mode code, and which were locked in memory by 386 | DOS-Extender under the control of the -REALBREAK switch at load time (please see Section 3.5).

INT 21h
AX = 251Ch

Free Physical Memory Pages

INPUTS:
BH = 0, if preserve page contents
= 1, if discard page contents
EDX = number of pages to free
If BL = 0:
ECX = linear address of first page to free
If BL = 1:
ES:ECX = pointer to first page to free

OUTPUTS:
If success:
Carry flag = clear
If failure:
Carry flag = set
EAX = error code
= 8 (memory error), if out of swap space
= 9, if invalid address range

The Free Physical Memory Pages system call is used by application programs which keep track of their own memory usage to optimize page replacement. It is made when the application knows that it will not be referencing a range of virtual pages for a long time. If this call is made when the virtual memory subsystem is not present, it simply returns without doing anything. See also function 252Bh subfunctions 7 and 8, which perform the same operations but may be easier to use.

This call frees up any physical memory pages allocated to the specified range of virtual pages, thus making them available for use by other virtual pages. If BH = 0, then the virtual page contents are preserved (i.e., the page is written to the swap file if it is dirty, as is normally done when replacing a page). If BH = 1, then the virtual page information is simply discarded, and the virtual page is marked to be zeroed, rather than read in from disk, the next time it is referenced. Thus, applications which don't care about the page contents can avoid the extra disk accesses necessary to write the page out if it is dirty and read it back in the next time it is referenced.

When the preserve page contents option (BH = 0) is specified, error 8 (memory error) may be returned. This can occur because when pages are flushed to disk, it may be necessary to increase the swap file, and it is possible to run out of swap space in the process. Note that this error never occurs if the -SWAPCHK MAX swap file increasing option is selected. However, if the default setting -SWAPCHK FORCE is selected, it is possible to run out of swap space, because this call results in some free (unused) physical memory pages, and because the swap file size algorithm employed by -SWAPCHK FORCE (please see Section 4.3) assumes that all available physical memory pages are utilized.

INT 21h **Get Memory Statistics**
AX = 2520h

INPUTS: DS:EDX = pointer to buffer at least 100 bytes in size
BL = 0, if don't reset 386 | VMM statistics
= 1, if reset 386 | VMM statistics

OUTPUTS: Carry flag = clear
100 bytes of buffer at DS:EDX filled in

Statistics about the application's memory usage, and about paging activity if the 386 | VMM subsystem is present, are returned. If BL = 1, then the virtual memory paging statistics are reset after they are read into the buffer. The 100-byte buffer pointed to by DS:EDX is filled in with statistics in the following format:

Offset	Size	Description
00h	DWORD	1, if the 386 | VMM virtual memory subsystem is present; 0, if not present
04h	DWORD	*nconvpg:* the total number of conventional memory pages available, alterable using system call 2525h or 2536h
08h	DWORD	reserved, always zero

Offset	Size	Description
0Ch	DWORD	*nextpg:* the total number of extended memory pages from all sources, that is, direct extended memory, VCPI, XMS, and special memory such as COMPAQ built-in memory. It is the sum of memory currently allocated plus what is reported as currently available. (Warning: This must be considered an approximate number if running under a multitasking environment such as DESQview or Windows. It must **not** be relied upon, because in a multitasking environment, another program might allocate it before you can.)
10h	DWORD	*extlim:* the extended memory page limit (can be set by the application with system call 2521h or 2536h, and defaults to *nextpg*, the total number of available extended memory pages. System call 2521h or 2536h disallows setting *extlim* higher than *nextpg*; but it can become higher after some other program consumes memory, reducing *nextpg*.)
14h	DWORD	*aphyspg*: the total number of physical pages allocated to the application
18h	DWORD	*alockpg*: the total number of locked pages owned by the application
1Ch	DWORD	*sysphyspg*: the total number of physical memory pages allocated to 386|DOS-Extender for system needs
20h	DWORD	reserved, contents undefined
24h	DWORD	the linear address of the beginning of the address space allocated to the application
28h	DWORD	the linear address of the end of the address space allocated to the application

Offset	Size	Description
2Ch	DWORD	the number of seconds since the last time 386|VMM statistics were reset
30h	DWORD	the total number of page faults since the last time 386|VMM statistics were reset
34h	DWORD	the number of pages written to the swap file since the last time 386|VMM statistics were reset
38h	DWORD	the number of reclaimed pages (page faults on pages that had previously been swapped to disk) since the last time 386|VMM statistics were reset
3Ch	DWORD	the number of code and data pages allocated to the application (i.e., the size in pages of the linear address space allocated to the application, not including any pages that are unmapped or mapped to a physical device address). 386|VMM will not permit this to exceed the size limit specified with -MAXPGMMEM.
40h	DWORD	the size, in pages, of the 386|VMM swap file
44h	DWORD	reserved, contents undefined
48h	DWORD	the minimum number of conventional memory pages below which the conventional memory block may not be shrunk (program pages placed in conventional memory with the -REALBREAK switch)
4Ch	DWORD	the maximum size, in pages, to which the swap file can be increased (controlled with the -MAXSWFSIZE switch)

Offset	Size	Description
50h	DWORD	*flags:* a 32-bit flags doubleword. The only bit currently defined is bit zero, which is set to 1 if a 386|VMM page fault is in progress (of interest, for example, to a critical error handler). Bits 1-31 are always cleared to zero.
54h	DWORD	*nfreepg*: the number of free physical pages guaranteed available by 386|DOS-Extender. The total number of physical pages currently consumed by 386|DOS-Extender on behalf of itself and the application is *aphyspg + sysphyspg + nfreepg*.
58h	DWORD	*navailpg*: the number of physical pages currently available, but not guaranteed to remain available in a multitasking environment. This number is always greater than or equal to *nfreepg*. It is equal to MIN(*nextpg, extlim*) - *aphyspg - sysphyspg*.

INT 21h **Limit Program's Extended Memory Usage**
AX = 2521h

INPUTS: EBX = limit, in pages, of physical extended memory which the application may use

OUTPUTS: If success:
Carry flag = clear
If failure:
Carry flag = set
EAX = error code
= 8, if insufficient memory or -NOPAGE switch used
EBX = maximum limit, in pages
ECX = minimum limit, in pages

This call sets the limit on the number of pages of physical extended memory which may be used by the application. See also function 2536h, which performs the same operation but may be easier to use.

The current extended memory limit is given by the *extlim* parameter obtained with system call 2520h, Get Memory Statistics. When the program starts up, it may use all the available extended memory on the computer. Under some circumstances, it is desirable to free some physical extended memory for use by another program. The program can continue to run normally after it reduces its usage of physical extended memory; it will just page more frequently because less physical memory is available in which to keep program virtual pages.

Physical memory is commonly freed before performing an EXEC to a child program, so that the child program will have physical memory available for its use. This system call should only be used when EXECing to a protected mode program, since real mode programs do not need to use extended memory. Note that when performing an EXEC to a protected mode child, it is also necessary to make sure that some conventional memory is available to load a second copy of 386|DOS-Extender. This can be done with system call 2525h, Limit Program's Conventional Memory Usage. After the child program has terminated, these calls are typically

used again to raise the limit back to the maximum value, so the application will run as efficiently as possible.

When this system call returns an error, it also returns the minimum and maximum values to which the limit may be set. The maximum value is always equal to the *nextpg* parameter returned by system call 2520h, Get Memory Statistics. The minimum value is usually zero, but may be higher if no physical conventional memory pages are available, or if the application program has locked many virtual pages in memory. This call may return an error if there is insufficient swap space available, since reducing the number of available physical pages causes more virtual pages to be kept on disk.

Using this call may result in considerable reshuffling of physical memory, and in some of the application's virtual memory being paged to disk, if the virtual memory subsystem is present. Locked pages are guaranteed to remain in memory, but they may be moved to different physical pages! Note that, if the application has aliased any locked virtual memory pages with system call 250Ah, Map Physical Memory, the page table entries for the aliased pages are not updated, even though the data may have been moved to a different physical page. Applications which alias virtual memory pages should never make this system call.

This system call can be made if 386|VMM is not present, but will only succeed if there are sufficient free pages available, since paging to disk is not an option without virtual memory.

INT 21h **Specify Alternate Page Fault Handler**
AX = 2522h

INPUTS: ES:EBX = address of alternate handler to invoke for invalid page faults

OUTPUTS: Carry flag = clear
ES:EBX = address of previous alternate page fault handler

This call specifies an alternate page fault handler to be given control when a page fault occurs on an unmapped virtual page, or on a page outside the linear address space allocated to the application program. The address of the previous alternate handler is returned.

The alternate page fault handler is given control by 386|VMM when a page fault occurs on a virtual page which is unmapped (not on disk). Normally, this is the result of a memory reference through an uninitialized pointer variable when the program is linked with the -OFFSET switch to create unmapped pages at the beginning of the program's virtual address space (please see Section 3.4). The default alternate page fault handler installed by 386|DOS-Extender simply aborts the application program and prints out a message identifying the program location where the error occurred.

An application program uses this system call to install its own alternate page fault handler. This is not normally recommended, since a page fault on an unmapped page usually indicates a program bug, and there is little which can be done about that at run time. However, this functionality is useful for someone writing a debugger program.

If 386|VMM is not present, this call is identical to using functions 2532h and 2533h to get and set the page fault handler vector.

INT 21h **Limit Program's Conventional Memory Usage**
AX = 2525h

INPUTS: EBX = limit, in pages, of physical conventional memory which the application may use

OUTPUTS: If success:
Carry flag = clear
If failure:
Carry flag = set
EAX = error code
= 8, if insufficient memory or -NOPAGE switch used
EBX = maximum limit, in pages
ECX = minimum limit, in pages

System call 2525h sets the limit on the number of pages of physical conventional memory which the application may use. Please see also function 2536h, which performs the same operation but may be easier to use.

The current conventional memory limit is given by the *nconvpg* parameter obtained with system call 2520h, Get Memory Statistics. When the program starts up, by default it is permitted to use all the available conventional memory which is not allocated for DOS or 386|DOS-Extender. If the -MINREAL and/or -MAXREAL switches are used, some conventional memory is left free at start-up time for other programs, and the protected mode application uses the rest. Under some circumstances, it is desirable to free some physical conventional memory for use by another program. The program can continue to run normally after it reduces its usage of physical conventional memory; it will just page more frequently because less physical memory is available in which to keep program virtual pages.

Physical memory is commonly freed before performing an EXEC to a child program, so that the child program will have physical memory available for its use. It may not be necessary to free up any memory with this system call if the -MINREAL and/or -MAXREAL switches are used, since,

in that case, there may already be sufficient free memory to load the child program. Note that some free conventional memory is required even when performing an EXEC to a protected mode child, because a second copy of 386|DOS-Extender will be loaded in conventional memory. After the child program has terminated, this call is typically used again to raise the limit to the original value, so the application will run as efficiently as possible.

When this system call returns an error, it also returns the minimum and maximum values to which the limit may be set. The maximum value is usually equal to the original limit when the program started up, but may be larger than the original limit if the -MINREAL and/or -MAXREAL switches are used to leave some free memory at load time. The minimum value is usually zero, but may be higher if no physical extended memory pages are available, or if the application program has locked many virtual pages in memory. It will also be nonzero if the application program used the -REALBREAK switch to make sure that part of the program is loaded in contiguous conventional memory. This call may return an error if there is insufficient swap space available, since reducing the number of available physical pages causes more virtual pages to be kept on disk.

Using this call may result in considerable reshuffling of physical memory, and in some of the application's virtual memory being paged to disk, if the virtual memory subsystem is present. Locked pages are guaranteed to remain in memory, but they may be moved to different physical pages! Note that, if the application has aliased any locked virtual memory pages with system call 250Ah, Map Physical Memory, the page table entries for the aliased pages are not updated, even though the data may have been moved to a different physical page. Applications which alias virtual memory pages should never make this system call.

This system call can be made if the virtual memory subsystem is not present, but will only succeed if there are sufficient free pages available, since paging to disk is not an option without virtual memory.

INT 21h **Memory Region Page Management**
AX = 252Bh

INPUTS: BH = subfunction code
= 0, to create unmapped pages
= 1, to create allocated pages
= 2, to create physical device pages
= 3, to map data file into allocated pages
= 4, to get page types
= 5, to lock pages
= 6, to unlock pages
= 7, to free physical memory pages, retaining data
= 8, to free physical memory pages, discarding data

If BL = 0
ECX = linear address of start of region
If BL = 1
ES:ECX = address of start of region
EDX = length of region, in bytes
[For other inputs, please see description of specific subfunctions.]

OUTPUTS: If success:
Carry flag = clear
[For other outputs, please see description of specific subfunction]
If failure:
Carry flag = set
[For error values, please see description of specific subfunction]

The Memory Region Page Management call is used to modify a range of pages within any segment owned by the application program. Pages can be in one of three states:

Allocated pages are normal memory pages which contain application code and data. If the program does not use 386|VMM, allocated pages are

always physical RAM memory pages. Under 386|VMM, allocated pages may be paged out to the swap file, and are loaded into memory when the application program references the page. Allocated pages are created when the application program is loaded, or when it allocates a segment or increases the size of a segment, or when it calls subfunction 1 of this system call.

Physical Device pages are used to access memory-mapped devices anywhere in the 4 GB physical address space of the processor. Physical device pages do not consume any physical RAM memory or VMM swap file resources. Physical device pages are created when the application program calls subfunction 2 of this system call, or calls INT 21h function 250Ah (Map Physical Memory).

Unmapped pages are pages which are marked not present and which unconditionally cause a page fault when referenced by the application (regardless of whether 386|VMM is present). By default, if a page fault occurs on an unmapped page 386|DOS-Extender will terminate the application. The application can install its own page fault handler for page faults on unmapped pages with INT 21h function 2522h, Specify Alternate Page Fault Handler. Unmapped pages are created at load time at the beginning of the application's code and data segment if the program is linked with the -OFFSET switch. The -OFFSET switch is normally used for detection of null pointer references by the application program. The application can use subfunction 0 of this system call to create unmapped pages anywhere within a segment, and can use system call 252Ch to add unmapped pages to the end of a segment. If the application then installs an alternate page fault handler, it can, for example, use unmapped page references to dynamically grow a data structure as needed.

The application memory region is defined by the starting address and length passed in to this call. Because the functionality provided by this call relies on the paging hardware of the processor, the requested action is always performed on units of 4 KB pages; i.e., the page-aligned memory region on which the action is performed must always begin and end on a 4 KB address boundary. For some subfunctions, the application memory region can begin and end on any byte boundary; other subfunctions require the caller to page-align the application memory region.

The following rules are applied when converting the application memory region to a page-aligned memory region:

- For all subfunctions, it is important that the requested action be performed on the entire application memory region. Therefore, partial pages at the beginning and end of the application memory region are always included in the page-aligned memory region.
- For some subfunctions, the requested action causes loss of data, and therefore should not be permitted to affect any memory outside the application memory region. For these subfunctions, the application memory region is required to be page-aligned, and error 9 (invalid memory region) is returned if the region does not start on a 4 KB boundary or its length is not a multiple of 4 KB.

Subfunctions 0-2 are not documented in this manual. Please see the *386|DOS-Extender Reference Manual* for documentation of those subfunctions.

INT 21h **Map Data File Into Allocated Pages**
AX = 252Bh
BH = 3

INPUTS:
If BL = 0
  ECX = linear address of start of region
If BL = 1
  ES:ECX = address of start of region
EDX = length of region, in bytes
DS:EDI = pointer to zero-terminated file name string
SI = DOS file access and sharing mode, as defined for INT 21h function 3Dh, Open File

OUTPUTS:
If success:
  Carry flag = clear
  EAX = 386|VMM file handle
  ECX = number of bytes mapped to file
    (= MIN(file size, region length))
If failure:
  Carry flag = set
  EAX = 2, if file error
    = 8, if memory error
    = 9, if invalid memory region
    = 129, if invalid parameters, or if 386|VMM not present
    = 134, if maximum number of 386|VMM file handles (4) already in use
  If EAX = 2 (file error)
    ECX = 1, if DOS error opening file
      = 2, if DOS error seeking in file
      = 3, if DOS error reading from file

EDX = error code returned by DOS, including (but not necessarily limited to):
= 2, if file not found
= 3, if path not found
= 4, if too many open files
= 5, if access denied
= 6, if invalid file handle
= 12, if file open access code invalid
= 32, if sharing violation

The Map Data File call is used to map a disk file into the virtual address space, and is only available if 386|VMM is present. Function 252Bh subfunction 0 (create unmapped pages) is first performed on the memory region to ensure that the pages are unmapped. Allocated pages are then created in the memory region, and pages are brought in from the data file by 386|VMM as they are referenced by the program. If the memory region is shorter than the file length, only the amount of data that fits in the memory region is available. If the file length is shorter than the memory region, the unused pages at the end of the memory region are made allocated pages are left as unmapped pages. The application memory region must be page-aligned.

If the file access is read-only, then modified pages in the memory region get written to the 386|VMM swap file, and the data file contents are never modified. If the file access is read/write, then modified pages get written back to the data file. Modified pages are written back to the data file only as 386|VMM selects them for replacement. All modified pages are written back to the file when the application program exits, or when the Close VMM File Handle call (INT 21h function 252Dh) is made. If all or part of the memory region is freed before closing the VMM file handle, some or all modified pages may not be written back to the data file. If a read/write data file has a partial page at the end of the memory region, the end of the page is zeroed each time the partial page is read in from the data file, so no data can be kept in the part of the page which is not within the file.

Read-only files require space in the swap file, so you can get an out-of-swap-space memory error when mapping a read-only file. Read/write files do not require swap space, because modified pages get written back to the data file rather than the swap file.

INT 21h
AX = 252Bh
BH = 4

Get Page Types

INPUTS:
If BL = 0
  ECX = linear address of start of region
If BL = 1
  ES:ECX = address of start of region
EDX = length of region, in bytes
DS:EDI = pointer to buffer, one WORD for each 4 KB page in the memory region

OUTPUTS:
If success:
  Carry flag = clear
If failure:
  Carry flag = set
  EAX = 9, if invalid memory region

The Get Page Types call is used to obtain the page type (allocated, physical device, or unmapped) for each of the pages in the application memory region. The page types are returned as one WORD per 4 KB page in the buffer at DS:EDI. The beginning of the application memory region is required to be page-aligned; there is no alignment requirement on the length.

The WORD of information returned for each page in the region contains the following data:

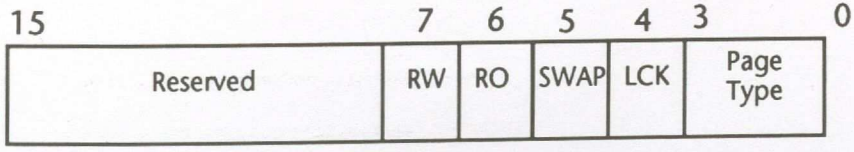


FIG C-1

Bit Mask	Description
000Fh	Page type code: 0, if unmapped page 1, if allocated page 2, if physical device page 3-15, reserved
0010h	Set if page is locked, cleared if unlocked
0020h	Set if page currently swapped to disk, cleared if in physical memory
0040h	Set if page is mapped to a read-only data file
0080h	Set if page is mapped to a read/write data file
FF00h	Reserved, always cleared.

INT 21h **Lock Pages**
AX = 252Bh
BH = 5

INPUTS: If BL = 0
ECX = linear address of start of region
If BL = 1
ES:ECX = address of start of region
EDX = length of region, in bytes

OUTPUTS: If success:
Carry flag = clear
If failure:
Carry flag = set
EAX = 8, if memory error
= 9, if invalid memory region

The Lock Pages call is used to lock allocated pages in physical memory. It is equivalent to INT 21h function 251Ah, but is easier to use because the 251Ah system call takes a length in pages and therefore requires the caller to calculate page alignment in order to figure out how many pages to lock.

The application memory region can begin and end on any byte boundary. It is permitted for the memory region to contain unmapped or mapped pages. Lock counts are not kept on pages; if a page is locked more than once, a single unlock call will unlock it.

INT 21h **Unlock Pages**
AX = 252Bh
BH = 6

INPUTS: If BL = 0
ECX = linear address of start of region
If BL = 1
ES:ECX = address of start of region
EDX = length of region, in bytes

OUTPUTS: If success:
Carry flag = clear
If failure:
Carry flag = set
EAX = 8, if memory error
= 9, if invalid memory region

The Unlock Pages call is used to unlock previously locked pages. It is equivalent to INT 21h function 251Bh, but is easier to use because the 251Bh system call takes a length in pages.

The application memory region can begin and end on any byte boundary. It is permitted for the memory region to contain unmapped or mapped pages, or allocated pages that have not previously been locked.

INT 21h **Free Physical Memory Pages, Retaining Data**
AX = 252Bh
BH = 7

INPUTS: If BL = 0
ECX = linear address of start of region
If BL = 1
ES:ECX = address of start of region
EDX = length of region, in bytes

OUTPUTS: If success:
Carry flag = clear
If failure:
Carry flag = set
EAX = 8, if memory error
= 9, if invalid memory region

This call is used to free physical memory pages, with the contents of the memory region preserved. It is equivalent to INT 21h function 251Ch (Free Physical Memory Pages) with the preserve page contents option set, but is easier to use because the 251Ch system call takes a length in pages.

The application memory region can begin and end on any byte boundary. This function has no effect if it is made when 386|VMM is not present. It is used by applications to improve performance, by informing 386|VMM that they will not be using a region of memory for a long time, so that 386|VMM can immediately free any physical memory allocated to the region for use elsewhere. Freeing physical memory differs from making pages unmapped because the pages are still allocated; the next time the application references them they will still be there.

INT 21h **Free Physical Memory Pages, Discarding Data**
AX = 252Bh
BH = 8

INPUTS: If BL = 0
ECX = linear address of start of region
If BL = 1
ES:ECX = address of start of region
EDX = length of region, in bytes

OUTPUTS: If success:
Carry flag = clear
If failure:
Carry flag = set
EAX = 8, if memory error
= 9, if invalid memory region

This call is used to free physical memory pages, with the contents of the memory region discarded. It is equivalent to INT 21h function 251Ch (Free Physical Memory Pages) with the discard page contents option set.

The application memory region is required to be page-aligned, so that no data outside the region is inadvertently lost. This function has no effect if it is made when 386|VMM is not present. It is used by applications to improve performance, by informing 386|VMM that they will not be using a region of memory for a long time, so that 386|VMM can immediately free any physical memory allocated to the region for use elsewhere. Any pages in the region that were previously allocated remain allocated after this call is made; however, unlike function 252Bh subfunction 7, the page contents are discarded and the page will be zeroed the next time it is referenced.

INT 21h **Close VMM File Handle**
AX = 252Dh

INPUTS: EBX = VMM file handle

OUTPUTS: If success:
Carry flag = clear
If failure:
Carry flag = set
EAX = 129, if invalid VMM file handle

This call is used to close an .EXP file or a data file that was kept open by 386|VMM so it can swap unmodified pages out of the file, reducing the space requirements for the swap file. The VMM file handle is obtained when the file is initially loaded with the Load File call (INT 21h function 2529h), the Load Program for Debugging call (INT 21h function 252Ah), or the Map Data File into Allocated Pages call (INT 21h function 252Bh subfunction 3).

If the file was a read/write data file, any modified pages are flushed to the data file before it is closed. You should **not** free any memory in the region mapped to a read/write file before making this call, or modified data may be lost (never be written back to the data file). For read only files or .EXP files, it is OK to free up the memory before making this call.

For all files (read/write data files, read only data files, .EXP files), all pages in the region mapped to the file are marked as unmapped pages by this call. For .EXP files, memory not paged out of the .EXP file (extra memory allocated at load time, or any memory subsequently allocated by system calls) is still present after this call is made. This call does not free up segments allocated to an .EXP file.

When the application program terminates, this call is automatically made for all open 386|VMM files. There is a maximum of 4 simultaneously open 386|VMM files, including the .EXP file for the application program.

If this call is made when 386|VMM is not present, it always returns success.

INT 21h **Get/Set 386 | VMM Parameters**
AX = 252Eh

INPUTS: DS:EBX = pointer to parameter buffer of 64 bytes
CL = 0, if getting parameters
= 1, if setting parameters

OUTPUTS: If success:
Carry flag = clear
Parameter buffer filled in, if getting parameters
If error:
Carry flag = set
EAX = 129, if bad parameter values

The parameter block has the following format:

Offset	Size	Description
0000h	DWORD	flags doubleword
0004h	DWORD	Scan period for updating page aging information, in milliseconds. Must be between 1000 and 2,147,483,648.
0008h	DWORD	Maximum size, in bytes, of linear address space to process on each page table scan for updating page aging information. Must be at least one megabyte.
000Ch	52 BYTES	Reserved for future expansion. Always returned as zero when getting 386	VMM parameters.

The flags doubleword has the following bit definitions:

Bit Mask	Description
00000001h	set if page fault logging enabled, cleared if page fault logging disabled. Page fault logging starts off enabled when the -PAGELOG switch is used. Page logging cannot be enabled at runtime if the -PAGELOG switch was not used. Only page fault records (record number 2) are not written to the page log file when page fault logging is disabled; all other record types are still written to the page log file as usual.
FFFFFFFEh	reserved for future expansion, always returned as zero when getting 386|VMM parameters.

For upward compatibility with future releases, the following steps should always be followed when setting 386|VMM parameters:

1. get 386|VMM parameters to initialize the values in the parameter buffer
2. modify only those parameters you wish to change in the buffer
3. set 386|VMM parameters.

An attempt to set 386|VMM parameters when 386|VMM is not present always returns success. An attempt to read 386|VMM parameters when 386|VMM is not present always returns with the entire parameter buffer zeroed.

INT 21h **Write Record to 386|VMM Page Log File**
AX = 252Fh

INPUTS: DS:EBX = pointer to data to write out
CX = size of data, in bytes

OUTPUTS: If success:
Carry flag = clear
If failure:
Carry flag = set
EAX = 133, if no page log file

The specified data is written to the 386|VMM page log file as a type 4 record. This call returns an error if the -PAGELOG switch was not used to turn on page logging.

INT 21h Minimize/Maximize Extended/Conventional Memory Usage
AX = 2536h

INPUTS: EBX = flags
bit 00000001h = 0, if modifying conventional memory usage
= 1, if modifying extended memory usage
bit 00000002h = 0, if minimizing memory usage
= 1, if maximizing memory usage
bit FFFFFFFCh = reserved, always clear to zero
If minimizing memory usage:
ECX = number of pages above minimum to set new limit to
If maximizing memory usage:
ECX = number of pages below maximum to set new limit to

OUTPUTS: If success:
Carry flag = clear
EAX = new limit, in pages, of extended/conventional memory usage
If failure:
Carry flag = set
EAX = 8, if memory error or -NOPAGE switch used
EBX = maximum limit, in pages
ECX = minimum limit, in pages

This call is used to restrict physical extended or conventional memory usage. It is similar to Limit Program's Extended Memory Usage (function 2521h) and Limit Program's Conventional Memory Usage (function 2525h). It can be more convenient than the latter calls because rather than

taking an absolute page limit, it takes a limit that is relative to the minimum or maximum possible limit, which means the caller does not need to ascertain what the acceptable range of limit values is before making the call.

INT 21h
AX = 253Ch

Reduce Size of Swap File

INPUTS: None

OUTPUTS:

Carry flag	= clear
EAX	= old size of swap file, in bytes
EBX	= reduced size of swap file, in bytes

This call attempts to reduce the size of the page swap file. It coalesces pages in the swap file and then truncates any unused pages from the end of the file. Depending on how many pages have to get moved to coalesce the file, this call can take a long time to execute (several seconds is not uncommon).

Normally, there are no unused pages in the swap file, so its size cannot be reduced. Unused pages appear in the swap file when the application program frees virtual memory, either by freeing a segment (function 49h), by reducing the size of a segment (function 4Ah), or by freeing pages within a segment (function 252Bh subfunction 0).

Standard memory allocators in compiler runtime libraries never free up virtual memory with one of the above system calls. You must either do your own memory management or modify the runtime library allocator to take advantage of this call.

Appendix D

System Programming Under 386|VMM

D.1 386|VMM Implementation

386|VMM maintains the data structures needed to implement virtual memory and also installs handlers for the page fault processor exception and the timer tick hardware interrupt. The page fault handler brings virtual pages into memory as needed, and the timer tick handler periodically updates the page aging information kept for use by the page replacement algorithm.

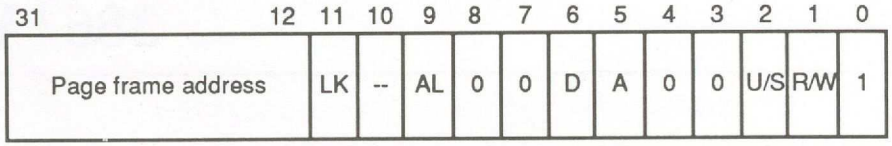
This appendix assumes a detailed knowledge of the paging functionality provided by the 80386 processor, which is specified in the Intel *80386 Programmer's Reference Manual*. Information contained in the Intel manual is not duplicated here.

D.1.1 Data Structures

The data structures maintained by 386|VMM are the 80386 page tables and a page table information segment which contains additional information about each virtual page. The system page tables (accessible through segment 00D0h) contain a four-byte entry for each virtual page. The page table information segment (segment 00D8h) also contains a four-byte entry for each virtual page. The index in these segments of the entry for a particular virtual page is obtained by shifting the linear address of the virtual page 12 bits to the right.

The information specified in this appendix, except where noted, is guaranteed not to change with future releases. The meaning of bits marked as reserved may change without warning in future releases of 386|DOS-Extender and/or 386|VMM.

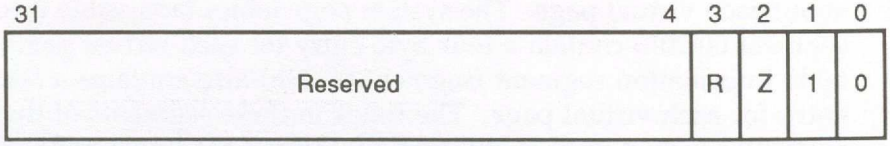
A page table entry for a virtual page which is currently in physical memory contains the following information:



Bit zero (the Present bit) is always set. The U/S (user/supervisor) and R/W (read/write) bits are used to limit access to the page and are disabled at all privilege levels except level three (to use this information, your program must run at privilege level zero, the most privileged level). The U/S and R/W bits are always set to one for consistency. The A (Accessed) bit is set by the 80386 when any type of access to the page occurs. The D (Dirty) bit is set when a write access occurs. All bits shown as zero are reserved by Intel for future expansion and are always set to zero.

Bits 9-11 are reserved for use by 386 | DOS-Extender. The AL (Allocated) bit is used to flag allocated physical memory pages, as opposed to pages mapped in with system call 250Ah, Map Physical Memory. The LK (Locked) bit is used to lock a page in memory, so that it cannot be swapped out. Bit 10 is reserved for use by 386 | DOS-Extender, and its meaning is *not* guaranteed to remain the same with future releases.

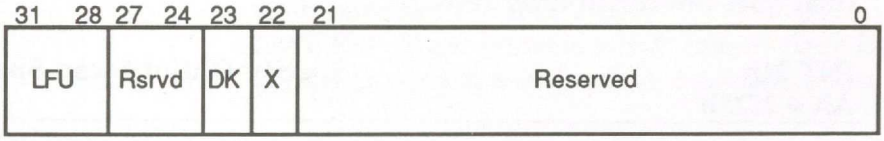
A page table entry for a virtual page which is currently on disk contains the following information:



Bit zero (the Present bit) is always cleared. Bit one (the Swapped bit) is always set to one, indicating that the page is swapped to disk. The Z (Zero) bit indicates whether the page is known to be all zeros. If the Z bit is cleared, the page is actually on disk. If the Z bit is set, the page is not on disk. The next time the page is referenced, it will be cleared to all zeros, instead of read in from disk, and then marked dirty. (Because it is not

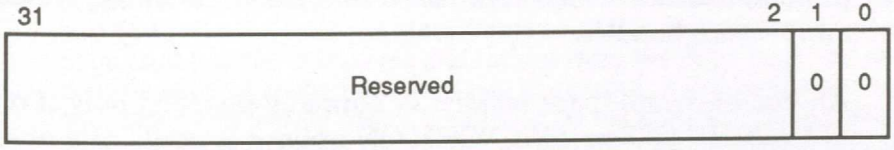
already on disk, it must be marked dirty so it gets written out to disk if it is replaced.) The Z bit is used to enhance performance when additional virtual memory is allocated with system calls, and also when system call 251Ch or 252Bh subfunction 8, Free Physical Memory Pages, is used to discard the contents of virtual pages. The R (Replaced) bit is used to indicate whether the page has been replaced; this is used for keeping statistics.

A page table information entry for a virtual page which is either in physical memory or on disk contains the following information:



The DK (Disk) bit is used to indicate whether the page is on disk; this must **always** be set for a swapped page which is not marked as Zero. However, it may or may not be set for a virtual page which is in memory, depending on whether it currently has a page allocated for it on the disk. The X (in eXp file) bit is set if the page is in the .EXP file or a mapped data file, and cleared if it is in the swap file. Virtual pages remain in the .EXP file until the first time they are modified; thereafter, they are kept in the swap file. The LFU bits hold a frequency count of the number of references to the page.

A free (unmapped) page table entry contains the following information:



Bit zero (the Present bit) and bit one (the Swapped bit) are both cleared, indicating the page is neither in physical memory nor on disk. The page table information entry for a free page table entry contains no information and is always zeroed.

D.2 Out-of-Swap-Space Handler

The out–of–swap–space handler is called by 386 | VMM when it runs out of space on the swap device during a page fault. After the handler returns, 386 | VMM again attempts to obtain a swap page; if the attempt fails again, a fatal error exit is taken.

By default, there is no out–of–swap–space handler installed. The application program can install one with the following system call. An out-of-swap-space handler can be installed by an unprivileged application (that does not run at level zero).

INT 21h **Specify Out-of-Swap-Space Handler**
AX = 2523h

INPUTS: ES:EBX = address of handler to call for out-of-swap-space condition

OUTPUTS: Carry flag = clear
ES:EBX = address of previous out-of-swap-space handler

This call specifies an out–of–swap–space handler to be called by the page fault handler, if it runs out of space on the swap device. The address of the previous handler is returned. An out–of–swap–space handler can be removed by passing a null address (0:0) in ES:EBX. If 386 | VMM is not present, this call always returns 0:0 as the address of the previous out–of–swap–space handler.

An out–of–swap–space handler is normally installed only if one of the -SWAPCHK OFF or -SWAPCHK ON options is used. Use of -SWAPCHK FORCE or -SWAPCHK MAX guarantees that you will never run out of swap space during a page fault, so long as you do not specify an excessively large value with the -CODESIZE switch. Please see Section 4.3 for details.

When the page fault handler runs out of swap space, it makes a FAR call to the out–of–swap–space handler . After the handler completes its processing, it returns control by executing a FAR return instruction. The handler can modify all general registers, but must preserve all segment registers. The linear address of the virtual page which was referenced to cause the page fault is passed as a single doubleword argument; the handler must not take any action which would affect that virtual page. Thus, the stack looks as follows when the handler gets control:

	DWORD offset (relative to SS) of interrupt stack frame (documented in the *386|DOS-Extender Reference Manual*) from page fault or INT 21h
	DWORD linear address which caused page fault
SS:ESP -->	QWORD return address from FAR procedure call

The handler must not cause any page faults during its processing, thus all of its code and data must be locked in memory. It can, however, make DOS or BIOS system calls. Since it is called in the context of the page fault handler task, it has close to one KB of stack space available.

The handler can do one of three things:

- It can attempt to create some free swap space so that the page fault handler can complete its processing and the program can continue to execute.
- It can resign itself to the program being aborted, and do whatever program cleanup is required before returning to the page fault handler, which will then take a fatal error exit and abort the program.
- It can free some swap space for page faults, and then change the return CS:EIP and SS:ESP in the interrupt stack frame so that it gets control again after the page fault handler completes and returns from the page fault. By obtaining control again after the page fault has completed, it can access unlocked code and data. This usually makes it easier to do full program cleanup and exit, although there is always the possibility that the program will access so many pages during cleanup that it will run out of swap space again.

Some possible ways for the handler to create more swap space so the program can continue are:

- delete some files on the swap device so the swap file size can be increased
- free up some pages in the swap file by calling the Free Physical Memory Pages function (251Ch) with the discard page contents option (BH = 1) set
- free up some pages in the swap file by freeing some virtual memory with INT 21h function 4Ah.

D.3 Page Replacement Handlers

Applications which have very unusual memory reference patterns that cause excessive thrashing with either of the two page replacement algorithms available under 386 | VMM can install their own page replacement handlers. When 386 | VMM needs a physical memory page, it calls the application's handler, which decides which virtual page is to be replaced in order to obtain a physical page. The idea is that since the application knows what its memory referencing patterns are, it can make more intelligent decisions than the generic algorithms employed by 386 | VMM.

Do not attempt to write your own page replacement handlers unless you have a thorough understanding of virtual memory and are experienced in writing system software. A bug in the page replacement handler can cause extremely subtle and mystifying errors in the execution of a program, and poorly written handlers can severely impact overall program performance. Page replacement handlers can only be installed by an application running at privilege level 0.

Page replacement handlers are installed by the following system call:

INT 21h **Specify Page Replacement Handlers**
AX = 2524h

INPUTS:
- ES:EBX = address of handler called to select a page for replacement
- FS:ECX = address of page aging handler, called periodically under the control of the -VSCAN switch
- GS:EDX = address of handler called each time a page is read into memory from the swap file

OUTPUTS:
- Carry flag = clear
- ES:EBX = address of previous page replacement handler
- FS:ECX = address of previous page aging handler
- GS:EDX = address of previous page read handler

The three handlers installed with this system call are called by 386|VMM under the circumstances described below. When installing page replacement handlers, all three handlers must be installed; it is not permissible to pass a null pointer for one of the handlers when the system call is made.

The handlers are called as FAR procedures, and they terminate by executing a FAR return instruction when their processing is completed. When the handlers execute, registers EAX, EBX, ECX, and EDX may be modified. The handlers must preserve the original values in all other registers. The code and any data used by the handlers **must** be locked in memory; the handlers must not cause another page fault under any circumstances. In addition, the handlers must not make any DOS or BIOS system calls. The stack size when the handlers are invoked is slightly less than one kilobyte.

The page replacement handler, specified by ES:EBX in the system call, is called by 386|VMM when it needs a physical memory page so that it can bring a virtual page into memory from the swap file. The job of the page replacement handler is to search the program's virtual address space and select a virtual page which is currently in memory, but which the handler

does not expect to be referenced by the application program for a long time. The selected virtual page is then written to the swap file if it is dirty. The virtual page is marked not in memory, and the physical page thus freed is reused. When the handler is called, two arguments are passed on the stack: the low and high linear addresses of the virtual address region used by the application program. This information is used by the handler to limit the number of page table entries it needs to search for an eligible virtual page. When the handler gets control the stack looks as follows:

	DWORD	1, if the page selected by the handler must be a page in the EXP file 0, if any swappable page is acceptable
	DWORD	one byte past ending linear address
	DWORD	starting linear address of virtual space
SS:ESP -->	QWORD	return address from FAR procedure call

The page replacement handler returns in the EAX register the linear address of the page it has selected to be swapped in the EAX register. If there is no page eligible for swapping, it returns the value zero.

The handler specified by GS:EDX in the system call is called whenever a virtual page is read into physical memory. The purpose is to allow the handler to initialize the four bit LFU field in the entry for the page in the page table information segment (selector 00D8h). This field is always initialized to zero when the virtual page is first allocated. It is not, thereafter, modified by 386 | VMM, including when the virtual page is paged to the swap file. The handler is passed the linear address of the virtual page which has just been read into physical memory as a single argument on the stack. When the handler gets control the stack looks as follows:

	DWORD	linear address of virtual page just read in
SS:ESP -->	QWORD	return address from FAR procedure call

The handler specified by FS:ECX in the system call is called periodically from within the timer tick handler installed by 386 | VMM. The period between calls is determined by the -VSCAN switch; the default value is every four seconds. The purpose of the handler is to update any virtual page aging information which is employed by the page replacement algorithm to select the best possible page to be replaced. The Dirty and

Accessed bits in the page table entry can be tested to see whether the page has been accessed and/or written to since the last time the bits were cleared by software. The Accessed bit can be cleared by this handler, if desired, so that the handler can always determine whether the page has been accessed since the last time it was called. The Dirty bit must **never** be cleared by **any** of these handlers. 386|VMM clears the dirty bit when a virtual page is first brought into physical memory, and it uses the dirty bit when a virtual page is replaced to decide whether it must be written to the swap file. The 4-bit LFU field in each entry in the page table information segment is available for use by this routine to keep page aging information. Note that since the page aging handler is called from within a hardware interrupt handler, it should complete its processing quickly. When the handler is called, two arguments are passed on the stack: the low and high linear addresses of the virtual address region used by the application program. This information is used by the handler to limit the number of page table entries it needs to scan when updating page aging information. When the handler gets control the stack looks as follows:

```
                DWORD   one byte past ending linear address
                DWORD   starting linear address of virtual space
SS:ESP -->      QWORD   return address from FAR procedure call
```

The code listed below illustrates how page replacement handlers should be written. This example code is included on the distribution disk for the development version of 386|VMM as a file named PFHNDLR.ASM. In this example, the page aging handler and the handler called each time the page is read into physical memory do nothing. The page replacement handler performs a circular scan of the virtual address space, and selects the next available in-memory page as the page to be swapped.

```
        .prot
        .xlisti
        include dosx.ah
        include hw386.ah

comment ~*********************************************

repl_hndlr(low_lin, high_lin, expf)
ULONG   low_lin;
ULONG   high_lin;
ULONG   expf;
```

```
tick_hndlr(low_lin, high_lin)
ULONG   low_lin;
ULONG   high_lin;

pgin_hndlr(linadr)
ULONG   linadr;

Description:
        Handlers for page replacement.  As simple as
        possible -- we simply do a circular scan
        always selecting the first page we find
        as the page to be swapped.
        Since we don't take any program activity into
        account when selecting a page to swap,
        neither the page aging handler called
        periodically, nor the handler called each
        time a virtual page is read into memory,
        need to do anything.

Calling arguments:
        low_lin         bottom of virtual region
        high_lin        top of virtual region
        expf            T ==> EXP page only
                        F ==> any swappable page
        linadr          lin addr of page just
                        swapped in

Returned values:
        0               if no pages to swap
        otherwise       linear addr of page to swap
~*****************************************************
;
; Since the linker creates a single program segment
; for a protected mode program, and both CS and DS
; point to it, we can put any global data
; needed in our code segment and still get to it
; from DS -- all we have to do is tell the assembler
; we can get to it by ASSUMEing that DS points to the
; code segment.  (Note, of course, that when the
; handler gets control DS is undefined and we have
; to set it to our data segment).
;
; The advantage of putting the data in the same segment
; as the handler code is that we know they will end
; up next to each other when the program is linked,
; which makes it simpler to lock our handler code and
; data in memory (if the code and data are in two
```

```
; different segments, they may not be adjacent so
; we would have to do two separate locks, one for the
; code and one for the data).
;
        assume  cs:text,ds:text
text    segment byte public use32 'code'

        public  hndlr_startp,hndlr_endp
        public  tick_hndlr,pgin_hndlr,repl_hndlr

;
; Start of handler code & data to be locked in
; memory
;
hndlr_startp label byte

;
; Global data needed by handlers.
;
swap_lin dd     0       ; linear addr of last page
                        ; swapped

tick_hndlr proc far     ; page aging handler
        ret
tick_hndlr endp

pgin_hndlr proc far     ; page read in handler
        ret
pgin_hndlr endp

repl_hndlr proc far     ; select page for replacement
;
; Stack frame
;
#EXPF   equ     (dword ptr 20[ebp])
#HIGH_LIN equ   (dword ptr 16[ebp])
#LOW_LIN equ    (dword ptr 12[ebp])

        push    ebp             ; Set up stack frame
        mov     ebp,esp                 ;
        push    es              ; Save modified regs
        push    fs                      ;
        push    ds                      ;
        push    esi                     ;
        push    edi                     ;
        mov     ax,14h          ; set up DS to our data
        mov     ds,ax                   ; segment
```

```
        mov     ax,SS_PGTAB     ; set up ES to point to
        mov     es,ax                   ; page table seg
        mov     ax,SS_PTINF     ; set up FS to page table
        mov     fs,ax                   ; info segment

        mov     eax,swap_lin    ; adjust last page
        cmp     eax,#LOW_LIN            ; swapped if out
        jb      short #fix              ; of range
        cmp     eax,#HIGH_LIN           ;
        jb      short #ok               ;
#fix:                                   ;
        mov     eax,#LOW_LIN            ;
        mov     swap_lin,eax            ;
#ok:

;
; Perform a circular scan of all page table entries
; in the virtual linear region, starting with 1 past the
; last page swapped, to find the first page we can swap.
;       EDI = page number of end of virtual region
;       ESI = page number of start of virtual region
;       EDX = current page number
;
        mov     eax,#LOW_LIN    ; branch if virtual linear
        cmp     eax,#HIGH_LIN           ; region null
        jae     #err                    ;
        mov     esi,#LOW_LIN    ; set ESI to start of
                                        ; range,
        shr     esi,PAGE_SHIFT  ; EDI to end
        mov     edi,#HIGH_LIN           ;
        shr     edi,PAGE_SHIFT          ;
        mov     edx,swap_lin    ; start with 1 past last
        shr     edx,PAGE_SHIFT          ; page swapped
        inc     edx                     ;
        cmp     edx,edi                 ;
        jb      short #loop             ;
        mov     edx,esi                 ;
#loop:
        mov     eax,es:[edx*4]  ; get pg tbl entry
        test    eax,PE_PRESENT  ; branch if not present
        jz      short #next             ;
        test    eax,PE_LOCKED   ; branch if locked
        jnz     short #next             ;
        test    eax,PE_ALLOC    ; branch if not allocated
        jz      short #next             ; (mapped)
        cmp     #EXPF,FALSE     ; if any page OK,
        je      short #done             ; we're done
```

```
        mov     eax,fs:[edx*4]  ; get pg tbl info entry
        test    eax,PTI_ONDISK  ; branch if not on disk
        jz      short #next             ;
        test    eax,PTI_INEXP   ; branch if in EXP file
        jnz     short #done             ;
#next:
        inc     edx             ; step to next page
        cmp     edx,edi                 ; & continue
        jb      #loop                   ;
        mov     edx,esi                 ;
        jmp     #loop                   ;
#done:

;
; OK, done with loop - return linear addr of page
; we found
;
        mov     eax,edx         ; return linear addr of
        shl     eax,PAGE_SHIFT          ; page to swap
        mov     swap_lin,eax    ; update last pg swapped

#exit:
        pop     edi             ; Restore modified regs
        pop     esi                     ;
        pop     ds                      ;
        pop     fs                      ;
        pop     es                      ;
        mov     esp,ebp         ; restore stack frame
        pop     ebp                     ; & exit
        ret                             ;

#err:
;
; Couldn't find any pages eligible for swapping
;
        xor     eax,eax         ; return no available
        jmp     #exit                   ; pages to swap
repl_hndlr endp

;
; End of handler code and data
;
hndlr_endp label byte

text    ends

        end
```

Index

E

F

G

H

I

L

M

N

O

P

Q

R

S

T

U

V

W

The people who wrote, edited, revised, reviewed, indexed, formatted, polished, and printed this manual were:

John Benfatto, Alan Convis, Noel Doherty, Lorraine Doyle, Bryant Durrell, Diego Escobar, Nan Fritz, Mike Hiller, Elliot Linzer, Bob Moote, Kim Norgren, Richard Smith and Hal Wadleigh.

60 Aberdeen Avenue, Cambridge, MA 02138
617-661-1510 Fax: 617-876-2972